ABSTRACT

ZHOU, WU. Repackaged Smartphone Applications: Threats and Defenses. (Under the direction of Xuxian Jiang and Peng Ning.)

Smartphone applications are rapidly growing in number and variety. These applications (or apps), typically organized in various app markets, can be conveniently browsed by users and simply tapped to install on a variety of mobile devices. In studying smartphone apps in these markets, we find a common “in-the-wild” practice of repackaging legitimate apps. This practice brings tremendous risks to app developers, mobile users, market operators, and the entire ecosystem. For example, malicious authors may attach destructive payloads to legitimate apps to infect unsuspecting users. Others may implant advertising code into popular apps to hijack ad revenue.

To better understand the extent and threats of repackaged smartphone apps, we conduct two systematic studies. First, we implement an app similarity measurement system called DroidMOSS that applies fuzzy hashing technique to effectively localize and detect changes from app-repackaging behavior. Using DroidMOSS, we conduct an initial sampling-based study on apps from six popular third-party markets. The study reveals a worrisome fact that 5% to 13% of apps in these markets are repackaged apps. Further investigation indicates that these repackaged apps are mainly used to replace existing in-app advertisements or embed new ones to hijack ad revenues. There are also cases where malicious payloads are implanted.

Not relying on sampling, the second study deals with all apps from the markets. Specifically, we employ a fast and scalable approach to detect piggybacked apps (the most serious category of repackaged apps). Realizing that attached payloads are not integral part of apps’ primary functionality, we propose module decoupling technique to partition apps into primary and non-primary modules. Observing that piggybacked app shares the same primary module as the original app, we develop a fingerprinting technique to extract meaningful semantic features into feature vector. We then construct a metric space and propose a fast search algorithm to efficiently and scalably detect piggybacked apps. A prototype named PiggyApp is implemented to study 84,767 apps collected from various markets. Results show the processing takes less than nine hours on a single machine and piggybacked apps constitute between 0.97% and 2.7% of all apps for these markets. Further investigation reveals a series of advertising libraries inserted into thousands of apps and a variety of malicious payloads implanted into dozens of apps. These results demonstrate the effectiveness and scalability of our approach.

To defend against app repackaging threat, we explore two different approaches. First, we propose a watermarking mechanism for Android apps as a deterrence mechanism. To embed and extract watermark automatically, we introduce manifest app, which can trigger different app functionality to exhibit the watermark within an extended Dalvik VM. The extracted watermark can be used as the proof of app ownership when repackaged app is identified. The second approach uses diversified intermediate
languages (other than Dalvik bytecode) to ship the code for various apps. Not knowing the instruction semantic, attackers will have difficulty in making meaningful modifications to the target app. To reduce performance overhead, we devise a lightweight in-app hooking mechanism to reuse Dalvik VM to interpret the new instructions. To eliminate developer’s intervention, we develop an automatic process to transform normal apps into protected form. To demonstrate the effectiveness of these two methods in defending against app repackaging, we analyze their robustness in resisting well-known attacks, and evaluated them against available tools. Evaluations show that both approaches introduce a small performance overhead adequate for daily usage.
Repackaged Smartphone Applications: Threats and Defenses

by
Wu Zhou

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

Computer Science

Raleigh, North Carolina

2013

APPROVED BY:

William Enck

Ting Yu

Xuxian Jiang
Co-chair of Advisory Committee

Peng Ning
Co-chair of Advisory Committee
DEDICATION

To my parents, wife and daughter.
**BIOGRAPHY**

Wu Zhou is originally from Hubei, China. He received his Bachelor of Engineering degree in Computer Application from the Tianjin University in China in 1996 and his Master of Engineering degree in Computer Application from Institute of Software, Chinese Academy of Science in 1999. After receiving his Master degree, he spent nine years working as a software engineer in the industry. Particularly, during his five years (2003-2008) at IBM China Development Laboratory, he lead the Linux on POWER toolchain project at IBM China and contributed tens of important patches to the GNU Debugger (GDB), which is the debugger having the largest user group worldwide. Since 2009, he spent another five years in USA to pursue his Doctor of Philosophy degree in Computer Science under the direction of Dr. Xuxian Jiang and Dr. Peng Ning. His primary research interests are in the area of systems security, such as operating system security, mobile security, cloud security, virtualization security and system software security. He will graduate with a Ph.D. degree in Computer Science from the North Carolina State University in December 2013. He recently joined Samsung Mobile as a Staff Software Engineer in the Mobile Communication B2B Security team and will continue in this role after graduation.
ACKNOWLEDGEMENTS

Working on the Ph.D. has been a wonderful and sometimes overwhelming experience for me. I would like to thank all those who have supported, encouraged and helped me in various ways during my graduate study.

First of all, I would like to thank my advisor, Dr. Xuxian Jiang, and my co-advisor, Dr. Peng Ning. They have given me invaluable support along the way and provided me a lot of insightful guidance for my work. Prof. Jiangs standard for excellence, enthusiasm in the work, and eye for details have improved me significantly as both a researcher and professional. I am very grateful to Prof. Jiang for offering me the opportunity to work on this topic, and kindly accommodating my personal constraints. I am very grateful to Prof. Ning for bringing me into NC State University and shaping my initial experience about security research. I am also very thankful to Prof. William Enck, and Prof. Ting Yu for their time and efforts serving in my PhD thesis committee. Their insightful comments and valuable advices helped a lot to improve this dissertation. I would also like to thank Prof. Douglas Reeves who graciously served as an examiner during my final defense presentation.

Second, I would like to thank National Science Foundation (NSF) and Army Research Office (ARO) for their funding support for my Ph.D. research. I am also grateful to all my co-authors for their helpful discussions and collaborations: Dr. Xinwen Zhang from Huawei Research Center, Dr. Shihong Zou from Beijing Univ. of Posts & Telecommunications, Dr. Ahmad-Reza Sadeghi from Technical University Darmstadt, Dr. Xiaolan Zhang, Dr. Glenn Ammons and Dr. Vasanth Bala from IBM T.J. Watson Research Center, and my colleagues in the Cyber Defense Lab at NC State, Zhi Wang, Michael Grace, Yajin Zhou and Ruowen Wang.

Third, I would like to give my thanks to my friends for their help during my Ph.D. study: Deepa Srinivasan, Lei Wu, Chiachih Wu, Minh Q. Tran, Kunal Patel, An Liu, Juan Du, Ahmed Moneeb Azab, Sangwon Hyun, Yao Liu, Emre Can Sezer, Attila Altay Yavuz, Xianqing Yu, Young-Hyun Oh, and Wenbo Shen.

Last but certainly not least, I deeply appreciate the everlasting support and encouragement from my family, particularly my wife, Hongli Dang, and our daughter, Annie Zhou. Also, I can never overemphasize the importance of my parents in shaping who I am and giving me the gift of education. Last, I am also very grateful to my parents-in-law for the kind support and care they have been providing to us all these years.
# TABLE OF CONTENTS

LIST OF TABLES ........................................................................ vii

LIST OF FIGURES ..................................................................... viii

Chapter 1 Introduction ................................................................. 1
  1.1 Background and Problem Overview ....................................... 1
  1.2 Dissertation Contributions ............................................... 3
  1.3 Terminology ...................................................................... 4
  1.4 Dissertation Organization .................................................. 5

Chapter 2 Related Work ............................................................. 7
  2.1 Repackaged Apps Detection ............................................... 7
  2.2 Apps Repackaging Defense ................................................. 8
  2.3 Smartphone Platform and App Security ............................... 10

Chapter 3 Detecting Repackaged Smartphone Apps in Third-Party Android Markets .......... 12
  3.1 Introduction ...................................................................... 12
  3.2 Design ............................................................................. 13
    3.2.1 Overview ................................................................... 14
    3.2.2 Feature Extraction ...................................................... 14
    3.2.3 Fingerprint Generation ............................................... 15
    3.2.4 Similarity Scoring ....................................................... 17
  3.3 Prototyping and Evaluation ................................................ 18
    3.3.1 Repackaged Apps in Alternative Markets ....................... 19
    3.3.2 False Negative Measurement ....................................... 23
  3.4 Discussion ....................................................................... 24
  3.5 Summary ........................................................................ 25

Chapter 4 Fast, Scalable Detection of Piggybacked Mobile Apps ........................................ 26
  4.1 Introduction ...................................................................... 26
  4.2 Design ............................................................................. 28
    4.2.1 Module Decoupling ...................................................... 29
    4.2.2 Feature Fingerprint and Representation ......................... 30
    4.2.3 Piggybacking Identification and Rider Analysis ............... 34
  4.3 Prototype and Evaluation .................................................. 34
    4.3.1 Evaluation Setup ........................................................ 35
    4.3.2 Module Decoupling Accuracy ....................................... 36
    4.3.3 Jaccard Distance Trade-Off ......................................... 36
    4.3.4 Piggybacking Detection .............................................. 37
    4.3.5 Rider Analysis .......................................................... 38
    4.3.6 Performance ............................................................. 41
  4.4 Discussion ....................................................................... 42
  4.5 Summary ........................................................................ 43
Chapter 5  AppInk: Watermarking Android Apps for Repackaging Deterrence  

5.1 Introduction ................................................................. 44  
5.2 Overview ................................................................. 46  
5.2.1 Problem Statement ..................................................... 46  
5.2.2 Software Watermarking ................................................. 47  
5.2.3 Challenges of Watermarking Android Apps ......................... 49  
5.2.4 Solution Overview ..................................................... 49  
5.2.5 Trust Model ............................................................. 50  
5.3 Design of AppInk ......................................................... 50  
5.3.1 Architecture ........................................................... 51  
5.3.2 Watermarking Code Generation ..................................... 52  
5.3.3 Manifest App Generation .............................................. 54  
5.3.4 Source Code Instrumentation ....................................... 57  
5.3.5 Watermark Recognizer ............................................... 58  
5.4 Implementation ......................................................... 60  
5.5 Analysis and Evaluation ................................................. 60  
5.5.1 Robustness Analysis .................................................. 61  
5.5.2 Evaluation with Repackaging Tools .................................. 63  
5.5.3 Performance Evaluation .............................................. 64  
5.6 Discussion ................................................................. 66  
5.7 Summary ................................................................. 67  

Chapter 6  Diversifying Intermediate Language for Anti-Repackaging on Android  

6.1 Introduction ................................................................. 68  
6.2 Background ............................................................... 70  
6.2.1 Android App and Dalvik VM ......................................... 70  
6.3 Design ................................................................. 72  
6.3.1 DIVILAR Overview .................................................... 72  
6.3.2 Virtual Instruction Selector ......................................... 73  
6.3.3 Bytecode Transformer ................................................. 74  
6.3.4 Virtual Instruction Interpreter ..................................... 75  
6.3.5 APK Packager .......................................................... 77  
6.4 Implementation ........................................................... 78  
6.5 Evaluation ................................................................. 79  
6.5.1 General Analysis ...................................................... 80  
6.5.2 Static Analysis ........................................................ 82  
6.5.3 Dynamic Analysis ..................................................... 83  
6.5.4 Virtualization Specific Analysis .................................... 84  
6.5.5 Performance and Memory Overhead Evaluation .................. 85  
6.6 Discussion ................................................................. 87  
6.7 Summary ................................................................. 88  

Chapter 7  Conclusion and Future Work ......................................... 90  

References ................................................................. 92
LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>The Numbers of Collected Apps from Official or Alternative Android Markets</td>
<td>19</td>
</tr>
<tr>
<td>3.2</td>
<td>Repackaged App Detection from Six Studied Third-party Android Markets</td>
<td>20</td>
</tr>
<tr>
<td>3.3</td>
<td>The Comparison of App Manifest Files from the Original and Repackaged Apps</td>
<td>21</td>
</tr>
<tr>
<td>4.1</td>
<td>The Dataset for PiggyApp Evaluation</td>
<td>35</td>
</tr>
<tr>
<td>4.2</td>
<td>Determining the Right Jaccard Distance</td>
<td>37</td>
</tr>
<tr>
<td>4.3</td>
<td>Piggybacking Detection Results</td>
<td>38</td>
</tr>
<tr>
<td>4.4</td>
<td>The Statistics of Piggybacked Ad Libraries</td>
<td>40</td>
</tr>
<tr>
<td>4.5</td>
<td>The Statistics of Piggybacked Malicious Payloads</td>
<td>40</td>
</tr>
<tr>
<td>5.1</td>
<td>App Execution Times and Extra Delays(in Seconds)</td>
<td>65</td>
</tr>
<tr>
<td>5.2</td>
<td>Extraction Times of Watermarked Apps</td>
<td>66</td>
</tr>
<tr>
<td>6.1</td>
<td>Example Mapping Rules for Opcode (in hex)</td>
<td>74</td>
</tr>
<tr>
<td>6.2</td>
<td>App Start Times and Extra Delays</td>
<td>86</td>
</tr>
<tr>
<td>6.3</td>
<td>Runtime Action Times and Extra Delays</td>
<td>86</td>
</tr>
<tr>
<td>6.4</td>
<td>Private Dirty Memory Sizes for Each App</td>
<td>87</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1.1</td>
<td>An Overview of the Dissertation</td>
<td>6</td>
</tr>
<tr>
<td>Figure 3.1</td>
<td>An Overview of DroidMOSS</td>
<td>13</td>
</tr>
<tr>
<td>Figure 3.2</td>
<td>Fuzzy Hashing for Fingerprint Generation</td>
<td>15</td>
</tr>
<tr>
<td>Figure 3.3</td>
<td>The Manifest File of a Repackaged App <em>com.tencent.qq</em></td>
<td>22</td>
</tr>
<tr>
<td>Figure 3.4</td>
<td>The Code Snippet of <em>execTask</em></td>
<td>23</td>
</tr>
<tr>
<td>Figure 4.1</td>
<td>The Overall System Architecture</td>
<td>28</td>
</tr>
<tr>
<td>Figure 4.2</td>
<td>An Example Module Decoupling Run</td>
<td>30</td>
</tr>
<tr>
<td>Figure 4.3</td>
<td>Triangle Inequality-based VPT Pruning</td>
<td>32</td>
</tr>
<tr>
<td>Figure 4.4</td>
<td>The Cumulative Distribution of Pair-wise Jaccard Distances</td>
<td>36</td>
</tr>
<tr>
<td>Figure 5.1</td>
<td>The Overall AppInk Architecture</td>
<td>50</td>
</tr>
<tr>
<td>Figure 5.2</td>
<td>Example Permutation Graph</td>
<td>52</td>
</tr>
<tr>
<td>Figure 5.3</td>
<td>Watermarking Code for the Permutation Graph in Figure 5.2</td>
<td>53</td>
</tr>
<tr>
<td>Figure 5.4</td>
<td>Example Manifest App Based on Robotium</td>
<td>54</td>
</tr>
<tr>
<td>Figure 5.5</td>
<td>User Interface Elements in App <em>NotePad</em>.</td>
<td>56</td>
</tr>
<tr>
<td>Figure 5.6</td>
<td>Event Flow Graph for <em>NotePad</em>.</td>
<td>57</td>
</tr>
<tr>
<td>Figure 5.7</td>
<td>Skeleton Code to Drive <em>NotePad</em>.</td>
<td>57</td>
</tr>
<tr>
<td>Figure 5.8</td>
<td>Work Flow of Watermark Recognizer</td>
<td>59</td>
</tr>
<tr>
<td>Figure 5.9</td>
<td>Shell Script to Drive Watermark Extraction</td>
<td>59</td>
</tr>
<tr>
<td>Figure 5.10</td>
<td>Watermark Embedding, App Repackaging &amp; Watermark Recognizing Snapshots</td>
<td>62</td>
</tr>
<tr>
<td>Figure 5.11</td>
<td>Execution Time of Watermarked App</td>
<td>65</td>
</tr>
<tr>
<td>Figure 6.1</td>
<td>Snip of Code Processed by Proguard</td>
<td>69</td>
</tr>
<tr>
<td>Figure 6.2</td>
<td>Architecture of Dalvik Virtual Machine</td>
<td>71</td>
</tr>
<tr>
<td>Figure 6.3</td>
<td>DIVILAR Architecture Overview</td>
<td>72</td>
</tr>
<tr>
<td>Figure 6.4</td>
<td>Function Hooking in DIVILAR</td>
<td>75</td>
</tr>
<tr>
<td>Figure 6.5</td>
<td>Components of Dalvik VM</td>
<td>76</td>
</tr>
<tr>
<td>Figure 6.6</td>
<td>Architecture of Dalvik VM after DIVILAR Loaded</td>
<td>77</td>
</tr>
<tr>
<td>Figure 6.7</td>
<td>Execution Flow of the Protected App</td>
<td>78</td>
</tr>
<tr>
<td>Figure 6.8</td>
<td>Exception When Applying <em>baksmali</em> on a Transformed App</td>
<td>81</td>
</tr>
<tr>
<td>Figure 6.9</td>
<td>Error Message When Applying Dare on a Transformed App</td>
<td>83</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Background and Problem Overview

Smartphones have recently gained significant popularity. A recent report revealed that in July 2011, the number of Android smartphone activations reached 550,000 per day [140]. The popularity of smartphones is a result of their mobility and, increasingly, a result of the large number and wide variety of feature-rich smartphone applications (or apps) available to mobile users. As an example of the exploding popularity of smartphone apps, as of December 2012 there were more than 600,000 apps on Google Play (the official Android market) [11]. These feature-rich apps extend the capability of smartphones by empowering users to browse, socialize, entertain, and communicate on the go with unprecedented convenience and interesting experiences, rather than limiting users to basic phone calls or simple text messages.

To allow mobile users to conveniently browse and install these smartphone apps, platform vendors created centralized markets, which include Apple’s App Store [69] and Google Play [72]. Developers can submit their apps to these centralized markets and make them available to thousands of users. Platform owners can also better control the quality of apps and block malicious apps to protect users. Meanwhile, a number of third-party markets were also created for various purposes (e.g., to meet regional or localization needs). Cydia [32] and Amazon AppStore [68] are two such examples that host thousands of apps for iPhone and Android users, respectively.

In studying mobile apps in these markets, we find a common “in-the-wild” practice of repackaging legitimate apps, and re-distributing the repackaged apps with a new signing key. As a technical method, app repackaging can be used for benign purposes. For example, Aurasium [148] uses app repackaging to intercept an app’s interaction with its underlying OS, aiming to enforce user-specified security policies for the app. And ADAM [152] uses app repackaging to tweak malware samples for the purpose of stress testing various Android anti-virus tools. However, app repackaging is more commonly used for surreptitious and malicious purposes, and brings tremendous risks to app developers, mobile users, market
operators, and the entire ecosystem. For example, malicious authors may attach destructive payloads to legitimate apps and advertise them in various markets to infect unsuspecting users. Others may attempt to implant advertising code into popular apps to hijack the ad revenue from the original developers.

Facing the widespread threat of app repackaging, we are still missing thorough understanding of the threats from the app repackaging and also effective security defenses. To better understand the extent of and threats from the app repackaging, and to help defend against app repackaging threats and foster a hygienic smartphone app ecosystem, we needed to answer the following questions: How serious is the overall app repackaging situation in the current app markets? What purposes are these repackaged apps used for? Can we systematically identify these repackaged apps? Can we detect these repackaged apps in a fast and scalable way in order to handle the fast influx of new app submissions each day? How can we defend against app repackaging and prevent their propagation on the Android platform? Note that answers to these questions can be very helpful in assuring users that a downloaded app is legitimate and does not contain any malicious payload. Moreover, developers are also protected since their intellectual property rights are not violated. In addition, market operators will be able to ensure that their markets are not populated with repackaged or trojanized apps. This dissertation is targeted to answer these questions. It is composed of four pieces of work.

To answer the first four questions, we conduct two systematic studies. First, we implement an app similarity measurement system called DroidMOSS (Chapter 3) that applies a fuzzy hashing technique to effectively localize and detect the changes from app-repackaging behavior. Leveraging DroidMOSS, we conduct an initial sampling-based study on apps from six popular third-party markets. The study reveals a worrisome fact: 5% to 13% of apps in these markets are repackaged apps. Further manual investigation indicates that these repackaged apps are mainly used to replace existing in-app advertisements or embed new ones to hijack ad revenues. There are also cases of repackaging in which malicious payloads are implanted into legitimate apps.

The second study (Chapter 4) does not rely on sampling. Instead, it deals with all apps from various markets. Specifically, the study employs a fast and scalable approach to detect piggybacked apps (the most serious category of repackaged app). Based on the fact that the attached payload is not an integral part of a given app’s primary functionality, we propose a module decoupling technique to partition app’s code into primary and non-primary modules. Recognizing that piggybacked apps share the same primary modules as the original apps, we develop a fingerprinting technique to extract meaningful semantic features and convert them into feature vector. We then construct a metric space and propose a linearithmic search algorithm to efficiently and scalably detect piggybacked apps. A prototype named PiggyApp is implemented and used to study 84,767 apps collected from various markets. Our results show that the processing of these apps takes less than nine hours on a single desktop machine. In addition, among these markets, piggybacked apps constitute between 0.97% and 2.7% of all apps. Further investigation reveals that a series of advertising libraries have been inserted into thousands of apps, and a variety of malicious payloads are implanted into dozens of apps. These results demonstrate the effectiveness and
scalability of our approach.

To answer the last question, we explore two different approaches. First, we propose a watermarking mechanism, named AppInk (Chapter 5), for Android apps. To embed and extract the watermark automatically, we introduce a new concept called manifest app, which is a companion for the app under protection and can trigger different functionality of the app to exhibit the watermark when the app is executed inside one extended Dalvik virtual machine (VM). Also our approach leverages automatic test generation to produce the manifest app so that the whole process needs minimum user intervention. To demonstrate the effectiveness of this method in preventing app repackaging, we analyze its robustness in defending against well-known attacks, and evaluate its resistance against open source repackaging tools. Our results show that AppInk is easy to use, effective in defending against current known repackaging threats on Android platform, and introduces small performance overhead.

Second, we propose DIVILAR (Chapter 6), a virtualization-based protection scheme to enable self-defense of Android apps against app repackaging. Specifically, it re-encodes an Android app in a diversified virtual instruction set and uses a specialized execute engine for these virtual instructions to run the protected app. However, this extra layer of execution may cause significant performance overhead, rendering the solution unacceptable for daily use. To address this challenge, we leverage a light-weight hooking mechanism to hook into Dalvik VM, the execution engine for Dalvik bytecode, and piggy-back the decoding of virtual instructions to that of Dalvik bytecode. By compositing virtual and Dalvik instruction execution, we can effectively eliminate this extra layer of execution and significantly reduce the performance overhead. We have implemented a prototype of DIVILAR. Our evaluation shows that DIVILAR is resilient against existing static and dynamic analysis, including those specific to VM-based protection. It also has a small performance overhead adequate for everyday usage (an average of 16.2% and 8.9% increase to the start time and run time, respectively).

1.2 Dissertation Contributions

Below, we highlight the main contributions of this dissertation towards understanding the threats of repackaged smartphone apps and developing effective defense solutions against app repackaging.

- **The first systematic study of app repackaging threat**  As a first step in understanding the extent of app repackaging threats, we implement an app similarity measurement system called DroidMOSS that applies a fuzzy hashing technique to effectively localize and detect changes from app-repackaging behavior. Using DroidMOSS, we conduct a sampling-based study on apps from six popular third-party markets. This study gives preliminary answers to questions about how prevalent app repackaging is and why apps are repackaged. It reveals a worrisome fact: 5% to 13% of apps in these markets are repackaged apps. Further investigation indicates that these repackaged apps are mainly used to replace existing in-app advertisements or to embed new advertisements.
to hijack ad revenues. There are also cases where malicious payloads are implanted.

- **A fast and scalable detection solution of piggybacked apps** Based on the fact that the attached payload is not an integral part of a given app’s primary functionality, we propose a module decoupling technique to partition an app’s code into primary and non-primary modules. Recognizing that piggybacked apps share the same primary modules as the original apps, we develop a feature fingerprinting technique to extract meaningful semantic features and convert them into feature vectors. We then construct a metric space and propose a linearithmic search algorithm to efficiently and scalably detect piggybacked apps. We implement a prototype named PiggyApp and use it to study 84,767 apps collected from various Android markets. Our results show that the processing of these apps takes less than nine hours on a single machine. In addition, among these markets piggybacked apps constitute between 0.97% and 2.7% of all apps. Further investigation reveals that a series of advertising libraries have been inserted into thousands of apps, and a variety of malicious payloads have been implanted into dozens of apps. These results demonstrate the effectiveness and scalability of our approach.

- **A dynamic graph based watermarking mechanism for app repackaging deterrence** By introducing the concept of companion app for an app under protection, we propose a dynamic watermarking mechanism for Android apps, which can be readily integrated into current app development practices. We evaluate it against two open source repackaging tools. The results show that our approach is effective in defending against commonly available automatic repackaging attacks, incurs small extra overhead to the app under protection, and also easy to use.

- **A diversified virtualization based protection for Android to thwart app repackaging** Through the design of a lightweight in-app hooking mechanism, our solution reuses the existing Dalvik VM and removes the need of the extra layer for virtualization instruction execution, which significantly reduces the extra performance overhead and make it adequate for everyday usage. Analysis shows that our protection is effective against common countermeasures, including static, dynamic analysis and particularly these specific to VM-based protection. Evaluation demonstrates that our virtualization based protection only incurs small overhead to app start and running time.

### 1.3 Terminology

**App market** An app market is a centralized marketplace used to host the smartphone apps submitted by developers. Through the app market, mobile users can conveniently browse smartphone apps and install them on their mobile devices.

**App repackaging** App repackaging is the behavior of repackaging mobile apps that belong to other authors and re-distributing the repackaged apps with a new signing key. It can be as simple as to re-sign
the disassembled app code, to modify the data or code in the original app logic, or to embed repacker’s own standalone functional (usually malicious) logic.

**Repackaged app and Piggybacked app**  A repackaged app is an app that has been disassembled, repackaged and re-distributed. As a special kind of repackaged app, a piggybacked app is an app in which non-original code piggybacks on the original app. Because of the introduction of non-original code, this type of repackaging is considered one of the most serious types of repackaged apps.

**Carrier app and Rider code**  The carrier app is the original app on which the non-original code piggybacks. Rider code is the non-original code that piggybacks on the carrier app. Typical rider codes include malicious payloads and ad libraries.

**Primary module and Non-primary module**  The primary module is the app component that performs the advertised functionality of the app (e.g., the game logic in game apps). The non-primary module is the app component that performs mainly supportive functionality (e.g., ad libraries, network connections, etc.)

**Watermarking and Software watermarking**  Watermarking is a technique used to prove the ownership of a specific artifact. It usually involves embedding a unique identifier into the target under protection in stealthy way. To prove the ownership, a watermark extracting process is usually applied on the target to recover the unique identifier. Software watermarking is used to identify the developer or owner of a specific software work, usually applied to combat software theft or plagiarism. It can leverage semantic-preserving transformation to encode some syntactic structure into the software such that the watermark can be extracted by statically looking into the code; or be encoded into a dynamic structure which have to be extracted by running the software in a specially design environment with specific input.

**Dalvik Bytecode and Dalvik Virtual Machine**  The main body of Android app is a special intermediate language, named Dalvik bytecode, which encodes the instructions to be executed to achieve app’s task. Like common Java bytecode, Dalvik bytecode has its own opcodes set and operands encoding [127], and needs a specially designed execution engine, named Dalvik virtual machine (VM), to run. Dalvik VM is an integral part of Android, whose responsibility is to decode Dalvik bytecode sequence and execute these encoded instructions in an interpretative or native way.

### 1.4 Dissertation Organization

Figure 1.1 shows an overview of this dissertation. Besides this introductory chapter (Chapter 1) and the next chapter on related work (Chapter 2), this dissertation presents four pieces of work in the following chapters. Chapter 3 presents the first systematic study of app repackaging threats (named DroidMOSS). Chapter 4 describes the design, implementation and evaluation of a fast and scalable solution for piggybacked apps detection (named PiggyApp). In Chapter 5, we describe the design, implementation and evaluation of AppInk – a watermarking based approach against app repackaging threats. Chapter 6 presents DIVILAR – an anti-repackaging solution using diversified instruction set. Finally, in Chapter
Figure 1.1: An Overview of the Dissertation

7, we conclude the dissertation and propose a few directions for future research.
Chapter 2

Related Work

In this chapter, we present some related work in the area of repackaged apps detection, repackaged apps defense, and smartphone platform and apps security in general.

2.1 Repackaged Apps Detection

The first area of related work includes research efforts to detect and analyze repackaged apps, and more general to measure software similarity. They are directly related to the DroidMOSS (Chapter 3) and PiggyApp (Chapter 4) systems of this dissertation.

Repackaged apps detection and analysis This category is the most closely related work to our DroidMOSS and PiggyApp systems. DroidMOSS is the first systematic study of app repackaging threats. Meanwhile, we notice an independent work from the App Genome project [81] that looks into the Android apps from two alternative China-based markets and reports that nearly 11% of their apps also available on the Android Market were found to be repackaged. However, the study does not disclose any methodology as well as technical details behind their findings. Based on their summary-style description, we observe that the results are only applied to those apps that are also available on the official Android Market. Our work – both DroidMOSS and PiggyApp – instead does not have this limitation by investigating apps we collected from six alternative geographically-scattered markets. Moreover, our study further looks into possible motivations behind repackaged apps and leads to unique insights (e.g., stealing or re-routing ad revenues – Section 3.3), which have not been reported by others.

More recently, there are a series of systematic studies about app repackaging threats [31,60,123]. For example, DNADroid [31] uses program dependency graph (PDG) to characterize Android app and compares PDGs between methods in app pair, showing resistance to several evasion techniques. Potharaju et al. [123] compares each app pair using different syntactic fingerprinting schemes, and can handle different levels of obfuscation used by the attacker. Juxtapp [60] collects static code features and represents them as bit vectors to improve the efficiency of pairwise comparison, it also supports incremental update
and distributed analysis. Similar to DroidMOSS, all these studies use pair-wise similarity measurement to study app repackaging threats, and thus has the same scalability issues. By proposing a new distance metric design and an associated nearest neighbor search algorithm, PiggyApp (Chapter 4) overcomes the scalability limitation from the pair-wise comparison as presented in the systems before, and achieves a better scalability with $O(n\log n)$ complexity.

**Software similarity measurement and searching** This category of related work includes prior efforts in measuring software similarity and detecting plagiarized code [64, 92, 132]. Among the most noted, MOSS [132] is designed to measure software similarity (at the source level) and has been widely used to detect plagiarism in college classes. DroidMOSS differs from it in two key aspects: First, DroidMOSS directly works at the Dalvik bytecode level without the source code access, which is required by MOSS. Second, both systems require the use of a sliding window to generate the fingerprint. However, MOSS uses it to generate a k-gram to directly compose the fingerprint, while DroidMOSS calculates the hash value to compare against a reset point to further localize repackaged changes.

Smit [64] leverages a similar Vantage Point Tree as leveraged in PiggyApp, but for a different purpose – large scale malware indexing and queries. In particular, by focusing on detecting malware variants, it does not have the need to further decouple internal modules, which is essential for our system. Similarly, BitShred [92] focuses on large-scale malware triage analysis by using feature hashing techniques to dramatically reduce the dimensions in the constructed malware feature space. After reduction, pair-wise comparison is still necessary to infer similar malware families. In comparison, PiggyApp focuses on a different problem, i.e., piggybacking detection among existing mobile apps, which necessitates module decoupling-like techniques to partition apps into primary and non-primary modules. Also, our linearithmic nearest neighbor search algorithm avoids the need of performing pair-wise comparison.

### 2.2 Apps Repackaging Defense

This area of related work includes research efforts on app repackaging protection and software protection in general. They are directly related to our app repackaging defense work in Chapter 5 (AppInk) and Chapter 6 (DIVILAR).

**Android app protection** To protect Android apps from piracy and foster the healthy development of Android app economy, Google has introduced several mechanisms. For example, Google recommends developers to leverage ProGuard [100] to optimize and obfuscate apps. Google also provides licensing verification library [71] to inquiry a server to verify if an app running on a mobile device is properly downloaded from Google’s app market. There are also attempts from other parties in this aspect. For example, Amazon and Verizon have introduced their own digital right management solutions for Android apps available in their app stores [122]. These mechanisms can increase the difficulty of reverse engineering Android apps, but are not strong enough to deter determined attackers from repackaging
through more laborious manual analysis. For example, open source tools are available to automatically crack these protections [103–105].

**Virtualization-based protection** Virtualization has long been applied to protect native code from reverse-engineering [52, 135, 136, 142]. Malware authors are also found to use virtualization to thwart detection or analysis. Meanwhile, countermeasures have been proposed to detect or reverse-engineer virtualization-based malware. For example, Rotalume can automatically reverse-engineer virtualization-based malware that employs a virtual machine with a fetch-decode-execute model [134]. To allow malware analyst to focus on the critical behaviors of malware, an inside-out approach is proposed to identify instructions that interact with the underlying operating system and other instructions that may affect those instructions [30]. Moreover, Ghosh et al. propose to use an attacking VM to replace the protecting VM to subject the protected software to analysis. It assumes that the protecting VM is not tightly bound to the execution environment [51]. DIVILAR (Chapter 6) leverages virtualization to protect Android apps from repackaging. An adversary of DIVILAR could apply these techniques against DIVILAR. In section 6.5.4, we have analyzed DIVILAR’s resilience against these approaches. Virtualization-based protection has also been applied to protect managed code as well (specifically MSIL) [7]. Their interpreters are implemented in managed code, causing performance overhead as high as 50 times to 3500 times. DIVILAR, also targeting managed code, achieves much better performance (Section 6.5) by combining virtual instruction and Dalvik bytecode execution.

**Software watermarking** Static watermarking embeds watermark into the code or data of applications [110,113], which usually involves syntax transformation and is vulnerable to semantic-preserving transformations. A variety of dynamic watermarking mechanisms have been proposed to overcome these attacks, including graph based [28,119], thread based [112], and path based watermarking [26]. AppInk (Chapter 5) does not claim any novel contribution in this aspect. We instead leverage existing dynamic graph based watermark to improve Android app’s capability in preventing and defending against common app repackaging attack.

**Java software protection** Android app is mainly written in Java. Due to its high-level expressiveness, Java code is relatively easier to be decompiled and reversed [124] compared with native code. To protect software written in Java, different solutions are pursued since its inception. One popular solution is to apply different levels of obfuscation to Java code, such as code or data layout obfuscation, control flow obfuscation, and string encryption [22,33,48]. Watermarking is also used to prove the ownership of Java code and to discourage Java software piracy [28, 110]. SandMark [27] is one popular research platform to study how well different obfuscation and watermarking mechanisms work in protecting Java software.

**Diversification for security** Similar to DIVILAR (Chapter 6), quite some researches apply the principle of diversity to enhance security. For example, address space layout randomization is a defense mechanism to prevent low-level memory-base exploits such as buffer overflow [133]. It randomizes locations of a program’s code or data so that an exploit cannot (easily) locate the vulnerability. It has
since been ubiquitously deployed in Linux [139], Windows [111], Android [125], and Mac OS X [35]. Instruction-set randomization is a closely related work. It diversifies a program’s instruction set to render injected attacking code ineffective (because they are encoded in a different instruction set) [14]. DIVILAR and other virtualization-based protection systems [7] adopt a similar idea to obfuscate a program in order to protect it from reverse-engineering. As software diversity becomes more popular, automated tools have been proposed to improve its performance. For example, Franz et al. use profile-guided optimization to reduce the overhead of software diversity [46]. This line of researches is orthogonal to our DIVILAR work and thus can be leveraged by us to further reduce the overhead.

2.3 Smartphone Platform and App Security

This area of related work covers a variety of projects [18, 25, 39–41, 43, 47, 55–57, 114, 117, 156–159] that have been undertaken to improve the smartphone platform and app security. Specifically, they can be loosely classified into three groups.

Analyzing the security properties for single app The first group of projects analyze a single app from various perspectives to identify possible security and privacy problems. For example, both TaintDroid [39] and PiOS [38] focus on the privacy leak problem, and respectively use dynamic taint analysis and static data flow analysis to infer potential privacy leaks. DroidRanger [158] instead combines both static permission analysis and dynamic footprint monitoring to detect malicious applications in existing Android markets. SCanDroid [47] examines an app’s manifest file to automatically extract a data flow policy, and then checks whether the data flows in the app are consistent with the extracted specification. Stowaway [43] studies a set of 940 apps and finds that about one-third are not following the principle of least privilege. Enck et al. [40] crafted a byte code decompiler to study 1, 100 popular Android apps for characterization. Apex [114], MockDroid [18], and TISSA [159] enhance the Android infrastructure so that users can better control the access to specific resources or permission at runtime themselves.

All these tools use static analysis, or dynamic analysis, or both techniques to infer some specific security or privacy properties for individual mobile app. In contrast, neither DroidMOSS nor PiggyApp rely on these expensive and complicated static or dynamic analysis techniques. Instead, DroidMOSS uses fuzzy hashing on instruction sequence and PiggyApp uses feature vectors to enable the rapid comparison of pairs of apps.

Studying the security of app interactions The second group is more closely related to DroidMOSS and PiggyApp, as it involves the interactions between apps. For example, one line of research [21, 25, 37, 44, 56, 117] studies the security risks caused by inter-application interaction. Among them, both ComDroid [25] and Saint [117] examine the interfaces third-party apps export in order to uncover possible unintended consequences. Woodpecker [56] focuses on a similar “capability leak” problem in Android firmware apps preloaded on the device. Numerous solutions to this problem have been pro-
posed. For example, Saint [117] further extends the Android framework to enforce a user-configurable inter-application policy. Felt et al. [44] proposes a mechanism called IPC Inspection that allows the framework to inspect the complete call chain that leads to a request for a dangerous feature. QUIRE [37] addresses the same permission delegation problem by proposing IPC call chain tracking to identify the provenance of these IPC requests and then enforce security checks. Bugiel et al. [21] use a run-time monitor to regulate communications between apps, to protect Android against confused deputy and colluding apps attacks.

While DroidMOSS and PiggyApp are concerned with the relationship between apps, they differs substantially from these systems in that they do not attempt to model the flow of information or control through an app; they are concerned with the similarity between two apps, not what they do or whether they may be tricked into doing something inappropriate.

**Studying the entire app markets for security purposes** PiggyApp presents a fast and scalable framework for repackaged app detection, and thus enables the handling of all apps from various markets. The third group of research [15, 16] focuses more broadly on entire app markets as well. For example, Stratus [15] explores the security problem of the whole app ecosystem composed of multiple markets, each of which has its own security policy, and proposes a new app installation method to retain the original single-market security semantics (e.g., kill switches or developer name consistency). While its focus is markedly different from ours, both directly concern the issues that face app markets today. Barrera et al. [16] uses a self-organization map to analyze 1,100 popular Android apps and identifies common usage patterns of permissions shared by different apps. Our approach also employs clustering techniques and extracts certain semantic features from the set of permissions an app requests. However, Barrera et al. uses them to visualize the relationships (in terms of requested permissions) among popular apps. PiggyApp instead makes use of them to build a VPT tree for efficient and scalable piggybacking detection.
Chapter 3

Detecting Repackaged Smartphone Apps in Third-Party Android Markets

3.1 Introduction

App repackaging, as a serious threat to the app ecosystem, was first publicly noticed on third-party Android markets in 2010 [82]. Because these alternative markets generally have a lot fewer constraints for app publishing, app repackaging is much more common on them than on the official Android market [83]. In fact, the first repackaged app on the official Android market was not publicly noticed until March 2011. In our initial study, we are motivated to systematically detect repackaged apps on third-party Android markets. In particular, we aim to shed some light on questions such as: How serious is the overall app repackaging situation in current third-party markets? What are these repackaged apps used for? Can we systematically identify these repackaged apps?

To answer these questions, we present a system called DroidMOSS to measure the similarity between two apps and use this information as the basis to detect repackaged apps. Specifically, given each app from a third-party Android market, we measure its similarities with those apps from the official Android Market. In order to handle a large number of apps in the official and alternative markets, we choose to extract some distinguishing features from apps, and generate app-specific fingerprints. Our fingerprint generation is based on a fuzzy hashing technique used to localize and detect the modifications repackagers apply to the original apps. After that, we calculate the edit distance to gauge how similar each app pair is. When the similarity exceeds certain threshold, we consider one app in the pair to be repackaged.

We have implemented a DroidMOSS prototype and used it to study six third-party Android markets worldwide, including two from the United States (with 6,296 apps), two from China (with 12,595 apps), and two from other countries.

1The name comes from an earlier system called MOSS [132] that is designed to measure software similarity and has been primarily used in detecting plagiarism in programming classes (based on source code submissions).
apps), and two from East Europe (with 4,015 apps). These apps were collected in the first week of March 2011 and they are measured against the 68,187 apps collected from the official Android Market in the same time frame. To perform a concrete analysis, we randomly picked 200 apps from each of these six markets, and measured their similarities with the total 68,187 apps in the official Android Market. From the resulting 81,824,400 pair-wise similarity scores, DroidMOSS systematically reports the repackaged apps. For each reported app, we perform a manual analysis and then calculate the false positive rate. Our results reveal that 5% to 13% of apps hosted in these six markets are repackaged apps (with false positive rates ranging from 7.1% to 13.3%). Also, we found that 13.5% to 30% of apps in these alternative markets are simply redistributed from the official Android Market. A further manual investigation indicates that these repackaged apps are mainly used to replace existing in-app advertisements or embed new ones to hijack ad revenues. We also identified a few serious cases with planted backdoors or malicious payloads in repackaged apps. These worrisome facts urgently call for a rigorous vetting process in the third-party app markets.

3.2 Design

To systematically detect repackaged apps in third-party markets, we have three key design goals: accuracy, scalability, and efficiency. Accuracy is a natural requirement to effectively identify app-repackaging behavior in current markets. However, challenges arise from the fact that the repackaging process might dramatically change the function naming or code layout in the repackaged app, which renders whole-app hashing schemes ineffective. Also, due to the large number of apps in various markets, our approach needs to be scalable and efficient. As a matter of fact, our current data set for app similarity measurement has 81,824,400 app pairs, which makes expensive semantics-aware full app analysis not feasible. Accordingly, in our design, we choose to collect syntactic instruction sequences from each app and then distill them for fingerprint generation. The generated fingerprints need to be robust in order to accommodate possible changes from app-repackaging behavior.
Assumption  In this chapter, we aim to uncover repackaged apps in current app markets and understand the overall situation. We focus on Java code inside Android apps without considering native code. One reason is that native code is harder for repackager to modify. Also our dataset shows that only a small number of (5\%) apps contains native code. Moreover, the apps from the official Android Market are assumed to be trusted and not re-packaged. There may exist exceptions to this assumption, but DroidMOSS is still helpful in distinguishing app pairs with repackaging relationship (Section 3.4). Finally, we assume that the signing keys from app developers are not leaked. Therefore, it is not possible that a repackaged app will be signed by the same author as the original one.

3.2.1 Overview

Repackaged apps share two common characteristics: First, due to the repackaging nature, the code base is similar between the original app and the repackaged app. Second, since the developers’ signing keys are not leaked, the original app and the repackaged app must be signed with different developer keys. DroidMOSS capitalizes on these two insights by extracting related features from apps and then discerning whether one app is repackaged from the other.

Figure 3.1 shows an overview of our approach. In essence, DroidMOSS has three key steps. The first step is to extract from each app two main features, i.e., instructions contained in the app and its author information. These two features are used to uniquely identify each app. After that, the second step is to generate a fingerprint for each app. The reason is that each app may contain hundreds of thousands of instructions. There is a need to significantly condense it into a much shorter sequence as its fingerprint (for similarity measurement). Finally, based on app fingerprints, the third step discerns the source of apps, i.e., either from the official Android Market or from the third-party markets, and measures their pair-wise similarity scores (so that we can detect repackaged apps). In the following, we examine each step in more detail.

3.2.2 Feature Extraction

Each Android app is essentially a compressed archive file, which contains the classes.dex file and a META-INF subdirectory. The classes.dex file contains the actual Dalvik bytecode for execution while the META-INF subdirectory contains the author information.

To extract Dalvik bytecode from classes.dex, we leverage existing Dalvik disassemblers (i.e., baksmali [3]). Initially, we use the Dalvik bytecode (with opcodes and operands) as the code feature directly. It turns out that it is not robust even for simple obfuscation that could just change some string operands (such as string names or hard-coded URLs). Because of that, we opt to make further abstraction by removing the operands and retaining only the opcode. The intuition is that it might be easy for repackagers to modify or rename the (non-critical) operands, but much harder to change the actual instructions. In the meantime, we also observe that apps intend to include various ad SDK libraries to fetch and display
ads. After being disassembled, these shared ad libraries unnecessarily introduce noise to our feature extraction. Fortunately, there are a limited number of them and our current prototype builds a white-list to remove them from the extracted code.

For the author information, the META-INF subdirectory contains the full developer certificate, from which we can obtain the developer name, contact and organization information, as well as the public key fingerprints. For simplicity, we map each developer certificate into one unique 32-bit identifier (or authorID). This unique identifier is then integrated into the signature for comparison.

### 3.2.3 Fingerprint Generation

For each app, our second step generates a fingerprint from the extracted code. A common way of achieving that is through hashing. Although hashing the entire code sequence of an app can uniquely determine whether two apps are the same, they are not helpful to determine whether two files are similar. The reason is simply because one minor modification will dramatically change the hashing value. From another perspective, calculating the edit distance between two given sequences is a well-known technique to measure their similarity. Unfortunately, it cannot be directly applied either. Considering each instruction sequence (of an app) could have hundreds of thousands of instructions, it will be very expensive to calculate one single edit distance between two apps, not to mention the large number of apps each needs to be paired and compared with others.

In DroidMOSS, we adopt a specialized hashing technique called *fuzzy hashing* [67]. Instead of directly processing or comparing the entire (long) instruction sequences, it first condenses each sequence into one much shorter fingerprint. The similarity between two apps is then calculated based on the shorter fingerprints, not the original sequences. Therefore, a natural requirement for fuzzy hashing is that the reduction into shorter fingerprints should minimize the change, if any, to the similarity of two
Algorithm 1 Generate the App Fingerprint

Input: Instruction sequence `iseq` of the app
Output: Fingerprint `fp`

Description: `wsize` - sliding window size, `rp` - reset point value, `sw` - content in sliding window, `ph` - the piece hash

1: `set_wsize(wsize)`
2: `set_resetpoint(rp)`
3: `init_sliding_window(sw)`
4: `init_piece_hash(ph)`
5: for all byte `d` from `iseq` do
6: `update_sliding_window(sw, d)`
7: `rh ← rolling_hash(sw)`
8: `update_piece_hash(ph, d)`
9: if `rh = rp` then
10: `fp ← concatenate(fp, ph)`
11: `init_piece_hash(ph)`
12: end if
13: end for
14: return `fp`

sequences.

To achieve that, we first divide the instruction sequence into smaller pieces. Each piece is considered as an independent unit to contribute to the final fingerprint. Therefore, if the repackaging process changes one piece, its impact on the final fingerprint is effectively localized and contained within this piece. For the rest pieces that are not changed, their contributions to the final fingerprint are still valid and persistent through the repackaging process, thus reflecting the similarity between the original app and the repackaged one. However, the challenge lies on the determination of the boundary of each piece. In DroidMOSS, we use a sliding window that starts from the very beginning of the instruction sequence and moves forward until its rolling hashing value equals a pre-selected reset point, which determines the boundary of the current piece. Specifically, if a reset point is reached, a new piece should be started. The concrete process is presented in Algorithm 1 and visually summarized in Figure 3.2.

For further elaboration, suppose a repackaged app has added a new instruction to invoke an external function. For simplicity, we assume the new instruction is inserted in the first piece of the instruction sequence (i.e., `piece 1` in Figure 3.2). Since our fuzzy hashing scheme uses a sliding window to calculate the rolling hash to determine the piece boundary, there are two possibilities about the placement of the new instruction in the first piece, either falling outside or inside the last sliding window. The former affects only the calculated hash value in the first piece while the rest pieces are intact. The latter changes the rolling hash value of the last sliding window (of the original `piece 1`). As a result, instead of stopping at the original boundary, we keep moving forward the sliding window until it hits the last sliding window.
in the second piece. In other words, it merges the first two pieces into one. Notice that it does not affect the determination of boundaries of the subsequent pieces. Therefore, for the final fingerprint generation, it only changes the hash values of the first two consecutive pieces. In either way, our scheme effectively localizes the changes.\(^2\)

In our design, to derive the fingerprint, we need to apply traditional hash function twice. The first is to calculate the hash value of each piece (after its boundary is determined) and the calculated hash values of all pieces are combined into the final fingerprint. The second is to calculate a hash value on the content of the sliding window, which is matched against the reset point. In our prototype, we use a prime number as the reset point to enhance the randomization or robustness of our scheme against possible repackaging attacks.

### 3.2.4 Similarity Scoring

Our first two steps are applied for each app regardless of its source. In the third step, we divide the apps into two groups, one from the official Android Market and one from alternative markets, and then calculate pair-wise similarity scores between the two. The similarity is based on the derived fingerprints, not the detailed (long) instruction sequence. Note that our fuzzy hashing scheme is deterministic in that if two apps from two groups are identical, the same fingerprints will be generated. In addition, it can also effectively localize the changes possibly made in repackaged apps.

Based on the above analysis, the similarity between the (shorter) fingerprints represents how similar their corresponding apps are. With that, our similarity scoring algorithm is to compute the edit distance between these two fingerprints, which is the number of minimum edit operations, including insertion, deletion and substitution of a single byte, needed to convert one fingerprint into another. The algorithm DroidMOSS adopts is a dynamic programming algorithm as presented in Algorithm 2. In particular, for two fingerprints \(fp1\) and \(fp2\) (with lengths of \(len1\) and \(len2\), respectively), we reserve a two-dimensional matrix (each value in the matrix is initialized to 0) to hold the edit distance between all prefixes of the first fingerprint and all prefixes of the second, and then compute the values in the matrix by flood filling the matrix. The distance between the two full strings will be the final value of the edit distance between the two fingerprints. The edit distance of any prefix subsequences of \(fp1\) and \(fp2\) can be derived from the minimum of three values: (1) \(\text{matrix}(i - 1, j) + 1\), which means to add one insertion operation in \(fp1\); (2) \(\text{matrix}(i, j - 1) + 1\), which means to add one deletion operation in \(fp2\); and (3) \(\text{matrix}(i - 1, j - 1) + \text{cost}\), which means to add one substitution operation between \(fp1\) and \(fp2\).

Based on the calculated edit distance, we can derive a similarity score between two fingerprints. The

\(^2\)Note that more advanced techniques could possibly mitigate our scheme. However, our prototyping experience and evaluation shows that they are not being used. From another perspective, the off-line nature of analyzing existing apps also makes our scheme easy to adapt and evolve.
Algorithm 2 Calculate the Edit Distance Between Two Apps

Input: Two fingerprints \( fp_1 \) and \( fp_2 \)

Output: Edit distance between \( fp_1 \) and \( fp_2 \)

1: \( \text{len}_1 \leftarrow \text{strlen}(fp_1) \)
2: \( \text{len}_2 \leftarrow \text{strlen}(fp_2) \)
3: initialize \( \text{two-dimensional_matrix} \text{matrix}(\text{matrix}, \text{len}_1, \text{len}_2) \)
4: for \( i = 0 \rightarrow \text{len}_1 \) do
5: for \( j = 0 \rightarrow \text{len}_2 \) do
6: if \( fp_1[i] = fp_2[j] \) then
7: \( \text{cost} = 0 \)
8: else
9: \( \text{cost} = 1 \)
10: end if
11: \( \text{matrix}[i, j] = \min(\text{matrix}[i - 1, j] + 1, \text{matrix}[i, j - 1] + 1, \text{matrix}[i - 1, j - 1] + \text{cost}) \)
12: end for
13: end for
14: return \( \text{matrix}(\text{len}_1, \text{len}_2) \)

The formula we are using is as follows:

\[
\text{similarityScore} = [1 - \frac{\text{distance}}{\text{max}(\text{len}_1, \text{len}_2)}] \times 100 \tag{3.1}
\]

If the calculated similarity score between two apps exceeds certain threshold and these two apps are signed with two different developer keys, our system reports the one not from the official Android Market as repackaged. The threshold selection affects both false positives and false negatives of our system. Specifically, a high threshold likely leads to low false positives but also high false negatives while a low threshold introduces high false positives but with low false negatives. During our experiments, we empirically found the threshold 70 is a good balance between the two metrics (Section 3.3).

3.3 Prototyping and Evaluation

We have implemented a prototype of DroidMOSS in Linux. In our prototype, the first step – feature extraction – is based on two open-source tools. Specifically, we use \textit{baksmali} [3], a popular Dalvik disassembler to reverse \textit{classes.dex} into an intermediate representation and then map it into Dalvik bytecode. A publicly available tool named \textit{keytool} [74], which is already a part of Android SDK, is used to extract the author information. To glue them together, we created a number of \textit{perl} scripts.

For the next two steps, we implement our own C programs for fingerprint generation and similarity scoring. For efficiency reason, our rolling hash function is based on the spamsum algorithm proposed by Andrew [8] (originally for spam detection) and the sliding window size in our prototype is 7. The
input to fingerprint generation is essentially those instruction sequences generated from the first step, while the output is used for similarity scoring. As mentioned earlier, the similarity scoring will also take into account the app author information: If two apps has the same authorID, we exclude them from repackaged app detection. If not, our prototype calculates the edit distance and derive the similarity score. The larger the score, the more similar the app pair.

To detect repackaged apps, we chose six popular third-party Android markets worldwide: two in US, two in China, and two in Eastern Europe\(^3\). For each market, we use a crawler to collect hosted apps. Our study is based on those apps collected in the first week of March, 2011. Meanwhile, we also collect more than sixty thousand apps from the official Android Market in the same time frame. The exact numbers of collected apps from official and alternative markets are shown in Table 3.1. For each alternative market, we also report the percentage of apps that are hosted in it but also have an identical copy in the official Android Market (i.e., the first category in Section 3.1). Table 3.1 shows our results.

### 3.3.1 Repackaged Apps in Alternative Markets

To perform a concrete study on the repackaged apps and measure the effectiveness of our approach, we randomly choose 200 samples from each third-party market and detect whether they are repackaged from some official Android Market apps. Specifically, for each chosen app, we measure its similarity score with each of these 68,817 ones inside the official Android Market. Among the calculated 68,817 similarity scores, we choose the highest one for manual investigation. Among the total 1,200 app pairs, we apply the threshold 70 to infer whether an app is repackaged or not.

Our results are shown in Table 3.2. The first column lists the name of these third-party markets; the second column indicates the number of repackaged apps detected by DroidMOSS out of the 200 samples; the third column shows the manual analysis results; the fourth one reports the corresponding repackaging rate; and the fifth column shows the 95% confidence interval (assuming that we have a

---

\(^3\)One domain is registered in Ukraine, but the resolved IP is actually located in US.
Table 3.2: Repackaged App Detection from Six Studied Third-party Android Markets (200 samples)

<table>
<thead>
<tr>
<th>Third-party Market</th>
<th># Repackaged Apps from DroidMOSS</th>
<th># Repackaged Apps from Manual Analysis</th>
<th>Repackaging Rate</th>
<th>95% Confidence Interval</th>
</tr>
</thead>
<tbody>
<tr>
<td>US1</td>
<td>24</td>
<td>22</td>
<td>11%</td>
<td>(6.8%, 15.2%)</td>
</tr>
<tr>
<td>US2</td>
<td>13</td>
<td>12</td>
<td>6%</td>
<td>(2.8%, 9.2%)</td>
</tr>
<tr>
<td>EE1</td>
<td>11</td>
<td>10</td>
<td>5%</td>
<td>(2.1%, 7.9%)</td>
</tr>
<tr>
<td>EE2</td>
<td>15</td>
<td>13</td>
<td>6.5%</td>
<td>(3.3%, 9.7%)</td>
</tr>
<tr>
<td>CN1</td>
<td>27</td>
<td>25</td>
<td>12.5%</td>
<td>(8.0%, 17.0%)</td>
</tr>
<tr>
<td>CN2</td>
<td>28</td>
<td>26</td>
<td>13%</td>
<td>(8.5%, 17.5%)</td>
</tr>
</tbody>
</table>

genuine random sample of the app data set) for this sampling study. For each market shown in the table, DroidMOSS reports that 5% to 13% of apps hosted on it are repackaged. Among the reported ones, we manually verify them and for each market we only find one or two false positives, demonstrating the effectiveness of our approach. By further looking into the false positive cases, we notice that one main contributing factor is that our white-list of ad SDKs or shared libraries is incomplete. Note that by iterating the process to complete the white-list, there is a room for our system to be further improved.

Overall, our experiments show that the repackaging rates range from 5% to 13% among these third-party markets. This is alarming as the repackaged apps seriously affect the entire smartphone app ecosystem. In the following paragraphs, we further look into individual repackaged apps and classify them into different categories for better understanding.

**Injecting New In-App Advertisements** In the first category, we observe new ad SDKs are added into the original app. Note that ad SDKs typically require adding a certain publisher identifier in the AndroidManifest.xml file, inserting layout description into the resource file, importing their own ad class files into the class directory, or even modifying the app bytecode. Recall that DroidMOSS considers ads as noise and thus filters them out for fingerprint generation and similarity scoring. With that, our system can easily spot them – as they share similar (or even identical) code sequences but are signed by different authors.

One example repackages a legitimate app *com.mmc.life49* by including the admob [70] SDK in the class hierarchy and adding a publisher identifier ADMOB_PUBLISHER_ID in AndroidManifest.xml. All the original bytecode remains intact. But some ad SDKs (e.g., wooboo [90] and youmi [151]) do require modifying existent class files in the original app to invoke ad-displaying code. Merely looking into the modified manifest file (or resource files) and newly added class files is not enough to identify this kind of repackaging. In general, the modification is applied on small part of the original code. DroidMOSS can readily localize such kind of modification and detect the repackaging.

**Usurping Existing In-App Advertisements** In the second category, we also observe repackaged apps where existing ad SDKs still remain, but the corresponding publisher identifiers have been replaced...
Table 3.3: The Comparison of App Manifest Files from the Original App and the Repackaged App

<table>
<thead>
<tr>
<th></th>
<th>Original Angry Birds (in Android Market)</th>
<th>Repackaged Angry Birds (in a US alternative market)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>&lt;manifest android:versionCode=&quot;142&quot;</td>
<td>&lt;manifest android:versionCode=&quot;142&quot;</td>
</tr>
<tr>
<td></td>
<td>android:versionName=&quot;1.4.2&quot;</td>
<td>android:versionName=&quot;1.4.2&quot;</td>
</tr>
<tr>
<td></td>
<td>android:installLocation=&quot;preferExternal&quot;</td>
<td>android:installLocation=&quot;preferExternal&quot;</td>
</tr>
<tr>
<td></td>
<td>package=&quot;com.rovio.angrybirds&quot; xmlns:android=&quot;......&quot;</td>
<td>package=&quot;com.rovio.angrybirds&quot; xmlns:android=&quot;......&quot;</td>
</tr>
<tr>
<td></td>
<td>&lt;application android:label=......&gt;</td>
<td>&lt;application android:label=......&gt;</td>
</tr>
<tr>
<td></td>
<td>......</td>
<td>......</td>
</tr>
<tr>
<td></td>
<td>&lt;meta-data android:name=&quot;ADMOB_PUBLISHERID&quot; android:value=&quot;a19c5b4602e&quot; &gt;</td>
<td>&lt;meta-data android:name=&quot;ADMOB_PUBLISHERID&quot; android:value=&quot;a19c5b4602e&quot; &gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;meta-data android:name=&quot;ADMOBInterstitial_PUBLISHER_ID&quot; android:value=&quot;a14ca2471ee08&quot; &gt;</td>
<td>&lt;meta-data android:name=&quot;ADMOBInterstitial_PUBLISHER_ID&quot; android:value=&quot;a14ca2471ee08&quot; &gt;</td>
</tr>
<tr>
<td></td>
<td>......</td>
<td>......</td>
</tr>
<tr>
<td></td>
<td>&lt;/application&gt;</td>
<td>&lt;/application&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;/manifest&gt;</td>
<td>&lt;/manifest&gt;</td>
</tr>
</tbody>
</table>

likely with the repackers’ identifiers. Note that each developer can sign up various ad networks to get his own app publisher identifier. The publisher identifier is assigned and used by an ad network to correctly distinguish user clicks or ad traffics and then return the resulting ad revenues. For example, Admob, one of the most popular ad networks in Android, uses two identifiers ADMOB_PUBLISHER_ID and ADMOBInterstitial_PUBLISHER_ID, whose values are assigned by Admob to the app developers during their enrollment. By repackaging apps with their own publisher identifiers, repackers can collect ad revenues from ad networks, resulting in a financial loss for the original app developers.

In our experiments, we found that one popular repackaged target is the Angry Birds app [143], whose package name is com.rovio.angrybirds. The vendor of this app (i.e., Rovio) does not charge for the download and installation. Instead it embeds certain ad SDKs (i.e., Admob) into this app to collect ad revenues. One repackaged Angry Birds DroidMOSS identified in a US market did not modify any code in the original app. Instead, the only modification is on the Admob-specific identifiers. Table 3.3 shows the comparison of two corresponding manifest files.

During our evaluation, we initially thought that applying a common Unix utility program, i.e., diff [66], on these two manifest files and their corresponding class files can easily identify such repackaging behavior. However, our experience indicates that repackers explored various unusual ways to
substitute publisher identifiers (Table 3.3). For instance, besides modifying the app manifest file, they may modify the string resource file instead without changing the bytecode at all. Fortunately, with its capability of effectively localizing the changes from repackaging behaviors, DroidMOSS can help detect them.

**Trojanizing Legitimate Apps with Malicious Payloads** In the third category, we also observe trojanized apps with malicious payloads. Our findings are consistent with recent reports about discovered Android malware [19]. Specifically, the added payloads can be used to conduct a variety of malicious activities, such as sending text messages to premium-rated numbers [99], downloading additional apps from the Internet [88], rooting the phone [93], and even registering the compromised phones as bots [82].

One example found by DroidMOSS is a repackaged app from `com.tencent.qq`, a popular instant messaging app. During our analysis, we found that the trojanized version requests more permissions as embodied in the first four lines of Figure 3.3. These permissions are requested to facilitate its wrongdoings. But having these added permissions is not sufficient to determine that one app is the repackaged version of another one. (Newer versions of an app may ask for more permissions than previous ones, and vice versa.) As a result, we need to further look into the code to collect additional evidences. In this particular case, the manifest file shows that a new receiver and a new service are added to the original app, and the receiver will be triggered when the system finished booting. Looking into the disassembled code, we know its purpose is to bootstrap a background service named `com.android.MainService`, whose code fetches and executes instructions from a remote server, effectively turning the compromised phones into bots. A further in-depth investigation of the code shows that the trojan app supports a number of commands, such as sending SMS messages to premium numbers, modifying the bookmarks of the built-in browsers, and downloading and installing additional apps onto the phones. All these actions are dispatched through a member function named `execTask` in `com.android.MainService`. 

---

**Figure 3.3**: The Manifest File of A Repackaged App `com.tencent.qq` (the listed receiver and service do not exist in the original app)
The function is invoked to check a command and control (C&C) server using a hard-coded URL (http://xml.XXX.com:8118/push/androidxml/?{parameters}) to fetch and execute commands. In Figure 3.4, we show a code snippet from this function that demonstrates how different payloads are called according to the command it receives.

Another example found by DroidMOSS is a repackaged app based on com.intsig.camscanner. A similar background service is needed in this case, but it is triggered in a different way. Instead of using a new receiver to trigger the service, the repackager directly modifies the main activity of the original app to achieve the same purpose. Our analysis also indicates that some obfuscation techniques are being adopted by repackagers to evade analysis and detection. It seems that these malicious payloads are getting more powerful and harder to be analyzed.\footnote{Our study also shows that there is a fourth category of apps. In this category, repackagers essentially decompose original apps and re-package them by signing with their own developer keys. One possible reason is that repackagers want to build their own reputation by providing benign, high-quality apps so that other users will trust them more when the time arrives for them to publish some bad or malicious apps.}

### 3.3.2 False Negative Measurement

While the above experiments focus on the understanding of overall repackaged apps in current third-party app markets, they also show the effectiveness of our system in having a small low false positives. Next, we measure the false negative rates of our system. Because there are no public list of known repackaged apps available for us to use, we prepare such a set by ourselves. Specifically, we first collect
those app pairs which have identical or similar package names, but are signed by different developer certificates. After that, we manually identify and confirm 150 repackaged apps as a test set to evaluate our system. As a result, DroidMOSS successfully reports 134 of them as repackaged, but misses 16 of them, implying a false negative rate of 10.7%. By examining those missed cases, we found two main reasons. (1) The first reason is that some repackager may add a large chunk of code into the original app. When the ratio between the added code and the original one is larger than certain threshold, the calculated fingerprints may differ a lot, leading to a small similarity score and causing a false negative. (2) The second reason is due to the fact our white-list is incomplete, which means some shared (ad) libraries are still contained in the sequence as noise. This added noise could result in considerable difference in the final fingerprints, thus leading to a miss in DroidMOSS.

To summarize, our experimental results show an alarming repackaging rate (ranging from 5% to 13%) among our current third-party markets. The repackaged apps are mainly used to replace existing in-app advertisements or embed new ones to “steal” or re-route ad revenues. We also identified a few cases with planted backdoors or malicious payloads among repackaged apps. The results call for the need of a rigorous vetting process for better regulation of third-party smartphone app markets.

3.4 Discussion

Our evaluation results show that our prototype can effectively detect repackaged apps. In this section, we further examine possible limitations in our system and explore ways for future improvement.

First, our current prototype assumes that the official Android Market contains legitimate (and original) apps. However, this may not be the case in practice. For example, it has been reported that even in the official Android Market, there may exist malicious apps [83] repackaged from other legitimate apps. Also, it is possible that an app (from a third-party market) might be an original one and the corresponding app from the official Android Market is actually repackaged. In either case, DroidMOSS is still helpful in distinguishing repackaged apps and answering the key questions that motivate this work.

Second, to discern any repackaged app, DroidMOSS depends on the existence of the corresponding original app in our data set. Due to various reasons, our current database is far from complete. For example, our current collection is comprised of those free apps and do not include paid apps in the official Android Market. As a result, we may miss some repackaged apps. Because of that, we have the reason to believe the overall repackaging rate is higher than we report in this chapter. From another perspective, this also indicates the need of continuously expanding our current data set with more comprehensive samples.

Finally, our prototype still experiences difficulties due to the use of shared libraries or ad SDKs for repackaged app detection. Specifically, our current approach uses a white-list approach that may not detect possible malicious changes to the ad SDKs or shared libraries. A systematic method to automatically identify shared libraries and detect abnormal changes could greatly improve our prototype.
3.5 Summary

In this chapter, we examine the problem of repackaged smartphone applications in current third-party markets, and have accordingly developed a prototype system called DroidMOSS to detect them. Our system adopts a fuzzy hashing technique to effectively localize and detect possible changes from app repackaging. We have applied our system to detect repackaged apps in six third-party Android markets and found that 5% to 13% of apps hosted in them are repackaged. Furthermore, we manually analyze those repackaged apps and our results show that apps are mainly repackaged to replace existing in-app advertisements (or embed new ones) to “steal” or re-route ad revenues, or even more seriously plant backdoors and malicious payloads. The results call for the need of a rigorous vetting process for better regulation of third-party smartphone app markets.
Chapter 4

Fast, Scalable Detection of Piggybacked Mobile Apps

Chapter 3 presents the first systematic study of app repackaging threats, leveraging the tool named DroidMOSS. However, the tool used in the study is not scalable due to its adoption of pair-wise similarity measurement. Having hundreds of thousands of apps at hand, we can only conduct a sampling based study. This chapter presents another systematic study based on a fast and scalable system – PiggyApp – for piggybacked apps detection, which is considered one of the most serious type of repackaged apps. With its high scalability, PiggyApp can deal with all apps from various app markets.

4.1 Introduction

Piggybacked app is a special kind of repackaged app, and involves injecting (new) rider code into the original apps. Repackaged app, however, may only make minor modifications to the original apps, including tweaking resource files or replacing constant strings for new language support. With the inclusion of new rider code, piggybacked apps pose greater security threats than other kinds of repackaged apps. In fact, a number of security alerts have been issued about the presence of piggybacked apps in various app markets. Specifically, these piggybacked apps embed malicious rider code into popular carrier apps, such as games and utility programs. Once installed, the rider code could perform a variety of malicious actions, such as sending text messages to premium numbers [89] and converting the infected phones into bots [88].

The similarity measurement based methods, as presented in Chapter 3 and in later research from other groups [31, 60, 123], can also be used to detect piggybacked apps. But all of these solutions use pair-wise comparison, in which the time complexity is $O(n^2)$ on the app set size. Considering the fact that there are already 600,000 apps on the official Android app market [11], and the fact that each day
there are around 1,000 new app submissions, we can see that the pair-wise comparison based solutions are not scalable enough to handle these large app data sets, and thus are not practical for real world adoption.

In this chapter, we propose a fast and scalable approach called PiggyApp to effectively detect piggy-backed apps in existing Android markets, including both official and unofficial ones. PiggyApp meets the need for scalability and timeliness by accommodating the fast influx of a large number of apps in existing markets, which dwarfs earlier approaches. Moreover, our system considers apps from different markets in the same manner, and thus enables the detection of piggybacked apps in the official Android Market (now part of Google Play).

Our approach is based on two main observations. First, in a piggybacked app, the rider code is relatively independent and does not tightly interweave, if any, with the primary functionality of the host app. Therefore, we propose a technique called module decoupling to effectively partition the app code into primary and non-primary modules. Each app has one unique primary module, which mainly implements the advertised functionality. Meanwhile, it may have a number of non-primary modules that are relatively standalone. Various support routines or libraries, advertisement packages, and mobile payment frameworks – as well as embedded rider code – fall into this category.

Second, a piggybacked app typically shares the same primary module as the original carrier app. Accordingly, we propose another technique called feature fingerprinting to extract certain semantic features (e.g., the requested permissions and the used Android APIs) embodied in the primary module. To facilitate this comparison and to meet our scalability requirements, we represent them as feature vectors, organize these feature vectors into a metric space, and then propose a linearithmic search algorithm (with $O(n \log n)$ time complexity – compared to the previous $O(n^2)$ complexity of pair-wise comparison) to detect piggybacked apps. From these piggybacked apps, we further derive the corresponding rider code and perform a systematic study of its functionality and purpose.

We have implemented a proof-of-concept prototype and used it to detect piggybacked apps in multiple Android markets worldwide, including the official market and six alternative ones: two from the US, two from Eastern Europe and two from China. Our study includes 84,767 apps and 68,187 of them come from the official Android Market. These apps are collected by taking a snapshot of the available apps on these app markets in the first week of March 2011. By running our system on a standalone desktop machine (with 4 cores and 8G of memory), it takes less than 9 hours to process all of these apps, which meets our scalability and timeliness requirements. The results show that 1.0% of apps in the official Android Market are piggybacked. For the rest alternative markets, the piggybacked apps vary from 0.97% to 2.7%. Further clustering analysis reveals a series of advertising libraries have been inserted into thousands of apps, and a variety of malicious payloads have been implanted into dozens of apps. These results demonstrate the effectiveness and scalability of our approach.
4.2 Design

In Figure 6.3, we show the overall architecture of our system. While piggybacked apps leverage carrier apps to entice users into downloading and installing them, the main purpose is to execute the attached rider code unnoticed. Notice that the rider code is relatively independent and should not closely interweave, if any, with the primary functionality of the carrier app. Accordingly, we propose module decoupling to first isolate the primary modules from existing apps. Moreover, as piggybacked apps still share the same primary module code base as the originals, we then propose to mainly compare primary modules to infer the piggybacking relationship between two apps.

In our system design, there are three competing goals: scalability, accuracy, and efficiency. Scalability is needed to accommodate the large number of apps in existing markets; accuracy requires our system to effectively detect piggybacked apps with few false positives and negatives; and efficiency imposes the need for our system to handle existing apps in a timely and resource-efficient manner. Specifically, to meet the scalability requirement, our system must improve upon the $O(n^2)$ time complexity of pairwise comparison in existing systems, such as DroidMOSS as presented in Chapter 3. To this end, we develop a feature fingerprinting technique that extracts semantic features from the primary module, including requested permissions and used Android APIs, and represents them as feature vectors. These feature vectors are used to construct a metric space from which we can efficiently identify similar apps using a linearithmic search algorithm ($O(n \log n)$ time complexity). By further examining the signing certificates and other non-primary modules of similar apps, we can effectively detect piggybacked apps as well as the related rider code.

In this work, we assume that piggybacking mainly occurs by adding Java code to a legitimate app, instead of native code. There are two main reasons: first, compared to native code, Java code is typically a more vulnerable target for piggybacking. A number of tools [19, 126] have been developed and can be readily misused for this purpose. Second, existing apps are still primarily written in Java, instead of C, which results in much less native code in existing apps. Considering the dataset used in this study, we find that only 5% of all apps contain native code. In addition, we assume that legitimate app developers do not disclose their private keys (for app signing) to others. Therefore, piggybacked apps will not share the same certificates as the original apps. Next, we detail each essential component in our system.
Algorithm 3 Agglomerative Clustering

Input: Program dependency graph (PDG) of an app
Output: A list of primary and non-primary modules

1: clusters = create.singleton.clusters(PDG)
2: while merge.able(clusters) do
3:     compute.coupling.between.each.pair(clusters)
4:     (c1, c2) ← select.the.most.coupled.pair(clusters)
5:     clusters ← merge(c1, c2, clusters)
6: end while
7: return clusters

4.2.1 Module Decoupling

An Android app is typically composed of multiple relatively independent modules. The primary module implements the main functionality, which is advertised to attract user downloads. Other non-primary modules may serve the primary module with support routines and utility libraries, but could also be completely independent (such as ad libraries). Within each module, either primary or non-primary, the code is tightly coupled or organized; between modules, the code is loosely coupled or even not related. (Some standalone apps may only contain one module – the primary module.) Without the access to the app’s source code, we resort to program comprehension techniques [10] to decouple internal modules within an app.

For a given app, our module decoupling process takes as input its classes.dex file and works in two main steps. First, based on the Dalvik bytecode, we build a program dependency graph (PDG). Within the graph, the node represents a Java class package that contains a number of Java class files declared within it. An edge connects two class packages if there exists an interaction or a dependency between these two packages. A weight is assigned to an edge to indicate how close these two class packages are connected. In our system, the edge essentially captures the following interaction or dependency relationship: class inheritance, package homogeny\(^1\), method calls, and member field references. Each of this relationship in general represents certain degree of coupling and our system will collect it and assign a weight. As the class inheritance relationship shows tighter coupling between two classes, we accordingly assign a higher weight to the edge than others that may simply indicate a single method call. Between two class packages, we use two cumulative weight values to unidirectionally sum all the edge weights from one to another.

Second, based on this program dependency graph, we use an agglomerative clustering algorithm (Algorithm 3) to group these class packages into different modules. To begin with, we initialize a number of singleton clusters one for each class package in the graph. After that, we repeat the process of

\(^{1}\)Two packages are homogeneous if they form a parent-child relation or share the same parent.
checking whether any two clusters can be merged and, if they can, merging the pair of clusters that have the largest cumulative weight values. Otherwise, we report the resulting set of clusters as the modules contained in the app. Note the mergeable condition (line 2) in the algorithm examines the remaining largest cumulative weight values between any cluster pair. In our prototype, we empirically choose a cut-off value (Section 4.3).

In Figure 4.2, we show an example run of the clustering algorithm on a piggybacked app (MD5: 09105460be466d0c024c37df8997b061). Initially, it has six modules com.rechild.advancedtaskkiller, com.google.ads, com.google.ads.util, org.json, com.android.root, and jackpal.androidterm. The figure shows the cumulative weight values between each pair. After the run, our algorithm effectively merges com.google.ads and com.google.ads.util and reports five remaining clusters as standalone modules.

Among the reported modules, we then determine which one is the primary module. In particular, we leverage the information in the AndroidManifest.xml file that declares various components of an app, including its activities, services, receivers, and content providers. Specifically, it specifies concrete classes that will be invoked to handle certain events or actions. A special one is ACTION.MAIN that represents the main entry point of the app. Accordingly, we choose the module that contains this class as a candidate for the primary module. Meanwhile, notice that the primary module tends to provide the main interface for users to interact with. Therefore, we also select the module with classes that handle most activities as the primary module candidate. If there are multiple candidates, we choose the most similar one by calculating the similarity of the module name against the app name in the manifest file. In our experiments, we do not encounter any piggybacked apps that change the app name to a completely unrelated one.

### 4.2.2 Feature Fingerprint and Representation

After module decoupling, we then generate feature fingerprints for the primary module. More specifically, feature fingerprints are supposed to distinguish the functionality of one primary module from
another. To this end, we extract various semantic features such as the requested permissions, the Android API calls used, involved intent types (which represent the way for inter-component or inter-process communication), the use of native code or external classes, as well as the authorship information (from the developer certificates in the META-INF directory). The intuition is that it is rare for two different modules to be coincidentally the same in all the above items. Notice that we include the developer information to exclude different apps authored by the same developer as there is no such need.

With the collected features, we then represent them into a vector where 0 and 1 respectively represent the absence and presence of certain feature in the primary module. After that, we organize these feature vectors (each representing an app) into a metric space and transform the problem of detecting piggybacked apps into a nearest neighbor searching problem. A naive approach for nearest neighbor searching is to perform pair-wise comparison and choose the one with the smallest distance. Considering the number of apps in current app markets, this approach is not scalable with its $O(n^2)$ complexity. That is also the main reason why we choose to construct a metric space from the extracted feature vectors. By exploiting its triangle inequality property [150], we can effectively prune irrelevant portion during the search and achieve an $O(n \log n)$ time complexity, thus accommodating the scalability challenge.

In the metric space construction, we use the following Jaccard distance between the primary module features of two apps as the distance metric:

$$Jaccard(F_A, F_B) = \frac{|F_A \cup F_B| - |F_A \cap F_B|}{|F_A \cup F_B|}$$  \hspace{1cm} (4.1)

where $F_A$ and $F_B$ represent the feature vector of primary module A and B respectively. Recall that they are vectors of binary values (0 or 1) to indicate whether one specific API call, permission, intent type or any external code loading behaviors occur in the module code. Formula 4.1 essentially calculates the ratio of disjoint features over the union of features present in these two modules to characterize how different they are. As shown in [137], the Jaccard distance satisfies the property of the triangle inequality that is being exploited to prune the irrelevant part of the search space.

In our system, we use a Vantage Point Tree (VPT) [150] to construct the metric space. Specifically, we first select a primary module as the root pivot $P$, measure the Jaccard distances between $P$ and all the rest of the modules, sort these modules in an ascending order of their distances to the pivot, and then divide them into a fixed number $N$ of balanced partitions, represented as $P_i$, $i = 1, 2, \ldots N$. At the pivot, the distance range associated with each partition $P_i$ is recorded, represented as $P_i.MIN$ and $P_i.MAX$. For each partition of the pivot, we will repeat this partitioning procedure to reduce its size to a manageable level.

To elaborate how the triangle inequality property enables efficient search pruning in the constructed VPT tree, we present in Figure 4.3 the partitions as concentric circles based on their distance ranges to the pivot. For an app query, suppose we discovered another app nearest neighbor with the minimum distance to query. The search space is then reduced to locate another app, say test, whose distance to
query is smaller than minimum. Due to the triangle inequality property in Jaccard distance, we have:

\[
\text{distance}(\text{query}, \text{test}) > |\text{distance}(\text{query}, \text{pivot}) - \text{distance}(\text{pivot}, \text{test})| \tag{4.2}
\]

If \(|\text{distance}(\text{query}, \text{pivot}) - \text{distance}(\text{pivot}, \text{test})| > \text{minimum}\) holds for any app test inside a partition, we can safely ignore this partition during the search. Thus we can use the following pruning conditions to skip any irrelevant partition \(P_i\) because it is not possible to find a shorter distance than minimum:

\[
\text{minimum} < \text{distance}(\text{pivot}, \text{query}) - P_i.MAX \tag{4.3}
\]

\[
\text{minimum} < P_i.MIN - \text{distance}(\text{pivot}, \text{query}) \tag{4.4}
\]

In Algorithm 4, we outline how nearest neighbor search works in VPT. It has two inputs: the query app query and the current root VPT node currentNode. During the search, we maintain two global variables, i.e., nearest_neighbor and the current minimum distance. If we reach a leaf node (lines 1 to 10), the algorithm simply computes the distances between the query app and each app stored in this leaf node. If any distance is smaller than current minimum value and they are also from different developers, we locate a closer distance and accordingly update minimum and nearest neighbor (lines 4 to 7). If we hit a pivot node (lines 11 to 24), the same procedure will be applied to the pivot app. Moreover, we will further examine each partition of this pivot node (line 18). If any of the pruning conditions in Formula 4.3 and Formula 4.4 are satisfied (line 19), this partition can be safely skipped; otherwise the nearest neighbor searching procedure will be recursively invoked on this partition (line
Algorithm 4 Nearest Neighbor Search in VPT: nearestNeighborSearch(query, currentNode)

1: if currentNode is a leaf node then
2: for each app in this leaf node do
3: if app and query not from the same author then
4: if minimum > distance(app, query) then
5: minimum = distance(app, query)
6: nearest_neighbor = app
7: end if
8: end if
9: end for
10: end if

11: if currentNode is a pivot node then
12: if pivot and query not from the same author then
13: if minimum > distance(pivot, query) then
14: minimum = distance(pivot, query)
15: nearest_neighbor = pivot
16: end if
17: end if
18: for each partition $P_i$ of this pivot code do
19: if minimum $<$ distance(pivot, query) $-$ $P_i$.MAX or
20: minimum $<$ $P_i$.MIN $-$ distance(pivot, query) then
21: continue
22: end if
23: nearestNeighborSearch(query, $P_i$)
24: end for
25: end if

22). To speed up the search, we can also initialize the minimum to a small number that indicates the acceptable level for piggybacking detection. This is possible because our previous module decoupling technique only retains primary modules for comparison while removing other non-essential ones (i.e., non-primary modules) as noise. Also, instead of only returning one nearest_neighbor, we can adjust the algorithm to report a list of apps that fall in a range of distance with the query app. In either case, the algorithm has the time complexity of $O(n \log n)$. 
4.2.3 Piggybacking Identification and Rider Analysis

By iterating through each app collected from an app market, our algorithm effectively reports a list of related apps which share similar primary modules and thus are candidates for piggybacked apps. To identify the exact piggybacking relationship, we take into account non-primary modules of related apps. Specifically, for each reported pair \( A \) and \( B \), we match their non-primary modules. If the non-primary modules of an app \( A \) are a strict sub-set of \( B \), any non-primary modules in \( B \), but not in \( A \), will be considered part of the rider code. Accordingly, we label the app with the rider code as the piggybacked app, and the other as the corresponding carrier app. If both apps have non-matched modules standing out, we choose to report them as a piggybacked pair, as we are not able to determine which one is piggybacked.

Besides determining the piggybacking relationship, we are also interested in what functionality is implemented in the rider code. While manual analysis in general cannot be avoided, our investigation shows that the same rider code may be injected into multiple piggybacked apps. Accordingly, we elect to cluster the detected rider code and group them for correlation. By doing so, we are able to identify several clusters whose members are very similar to each other. In our prototype, we choose to reuse the previous algorithm and build another VPT tree (Section 4.2.2) only for these identified riders. Our experience shows that the number of rider-related non-primary modules is one magnitude smaller than that of apps, which allows us to select a smaller distance (as the range parameter). As to be shown in Section 4.3.5, such clustering quickly exposes several clusters with the same rider code piggybacking on a number of carrier apps.

4.3 Prototype and Evaluation

We have implemented a prototype of PiggyApp in Linux. In our prototype, the first component – module decoupling (Section 4.2.1) – is implemented by extending the open source Dalvik disassembler baksmali [3] (with an additional 1926 lines of Java code) to generate the program dependency graph (PDG) and then isolate primary modules from other non-primary modules. When generating the graph, we assign weights 10, 10, 2, 1 to edges representing class inheritance, package homogeny, method calls, and member field references, respectively. The cut-off value for the mergeable condition (Algorithm 3) is empirically set to 5, which works well in practice (Section 4.3.2).

The second component – feature fingerprint and representation (Section 4.2.2) – extracts the semantic features of 32,011 APIs, 136 permissions, 122 intent types, 180 content provider features, and 2 additional code loading features, which essentially condense each app into a feature vector of length 32,451. These feature vectors are then organized into a Vantage Point Tree (VPT) that is implemented in 2,731 lines of C code. For search efficiency, we set the number of partitions at each pivot node to 3. To strike a good balance between accuracy and efficiency, we select Jaccard distance 0.15 for the
<table>
<thead>
<tr>
<th>Market</th>
<th>Total Number of Apps</th>
</tr>
</thead>
<tbody>
<tr>
<td>slideme (US1)</td>
<td>3108 (29.8%†)</td>
</tr>
<tr>
<td>freewarelovers (US2)</td>
<td>3188 (13.2%†)</td>
</tr>
<tr>
<td>eoemarket (CN1)</td>
<td>8261 (30%†)</td>
</tr>
<tr>
<td>goapk (CN2)</td>
<td>4334 (13.5%†)</td>
</tr>
<tr>
<td>softportal (EE1)</td>
<td>2305 (19.6%†)</td>
</tr>
<tr>
<td>proandroid (EE2)</td>
<td>1710 (20.2%†)</td>
</tr>
<tr>
<td>Official Android Market</td>
<td>68187</td>
</tr>
</tbody>
</table>

similarity measurement of two primary modules. We will detail how we choose this Jaccard distance in Section 4.3.3.

The third component – piggybacking identification and rider analysis (Section 4.2.3) – is implemented in 611 lines of Python code. Basically, it scans the list of candidate app pairs reported from the second component, fetches the non-primary modules of related apps, determines the piggybacked apps, and exposes the rider code. In our implementation, we re-target the second component to organize the rider code with one exception: no author information is needed to constrain the nearest neighbor search.

To evaluate the scalability and efficacy of our system, we use it to detect piggybacked apps in a dataset with 84,767 apps collected from seven different app markets. In the following, we first present our evaluation setup and then assess the accuracy of our module decoupling technique. After that, we determine the Jaccard distance for similarity measurement and report the detection results, including the analysis of uncovered rider code. Finally, we report the performance overhead.

### 4.3.1 Evaluation Setup

In Table 4.1, we summarize the collected apps in our dataset. Basically, our crawler takes a snapshot of the available apps from seven different Android markets in the first week of March, 2011. In total, the dataset contains 84,767 distinct apps: 68,187 of them appear in the official Android Market [72] and the rest come from other six popular third-party markets: two in the US, two in China, and two in Eastern Europe. We highlight that we analyzed all these 84,767 apps in this data set, which is made possible by our scalable analysis framework. Earlier systems such as DroidMOSS [155] employ pairwise comparison, which is not scalable and can only work on limited samples (e.g., 200 in [155]).

For each app in our dataset, our system extracts 32,451 semantic features and presents them in a vector. In total, our system produces 84,767 feature vectors. To understand the distribution of each app pair distance, we randomly select 2,000 and 4,000 samples and measure their distances with all other apps in the dataset. The results are shown in Figure 4.4 (with the y-axis in the log-scale). As expected,
most apps are not similar to each other, which is reflected by the fact that a majority of distances (around 99.4%) are larger than 0.8 (the largest possible distance is 1). Also, there are a small fraction (0.06%) of apps whose distances fall below distance 0.2, suggesting most piggybacked apps are located in this range (Section 4.3.4). This distribution is helpful to create a balanced VPT tree and leads to efficient nearest neighbor search.

### 4.3.2 Module Decoupling Accuracy

Module decoupling is an essential component, which affects both the accuracy and efficiency of our system. To concretely evaluate its effectiveness, we randomly choose 200 samples from our dataset, apply the module decoupling algorithm (Algorithm 3), and then manually verify the decoupling results. As with the module decoupling process, verification involves two main aspects. The first one is to determine whether these apps are decoupled into the correct modules. Our results show that 193 apps (96.5%) are correctly decoupled. The second aspect is to determine whether primary modules are correctly labeled. For the correctly decoupled 193 apps, our system identifies 178 primary modules (92.2%). We further examine the 15 mis-labeled apps and find that most cases, especially game- and social network-related apps, use feature-rich engines or libraries (e.g., Scoreloop [87] and Openfeint [118]) for GUI rendering, user interaction, and virtual currency support. They are generally considered as the main functionality of an app, but are implemented as supporting frameworks and shared among related apps. As these mis-labeled cases are rare and usually come from a limited number of special-purpose SDKs, we choose to apply a quick patch to our prototype by using a short white-list.

### 4.3.3 Jaccard Distance Trade-Off

Next we present our experiments to determine the proper Jaccard distance in our study. As it measures the overlapped semantic features, our feature fingerprint is by design robust to existing code obfuscation.
techniques [2]. Moreover, it provides a “tuning knob” to adjust the trade-off between accuracy and efficiency. Specifically, a larger distance will likely tolerate more disjoint features between two apps, which has the benefit of reducing false negatives but at the cost of increasing false positives. A smaller distance leads to more false positives but less false negatives. As a general rule of thumb, if two apps have a Jaccard distance greater than 0.3, we consider the possibility of having a piggybacking relationship to be very low.

In our study, we aim to achieve a lower threshold to improve the search efficiency while still obtaining sufficient accuracy. To this end, we choose 4,000 random samples and use a series of Jaccard distances to measure their accuracy. Specifically, for each distance, we calculate the true positives and false positives by examining each reported pair (as candidate piggybacked apps). The results are shown in Table 4.2. The experiments clearly indicate the Jaccard distance 0.15 as the threshold. In particular, the larger distances (> 0.15) detect two more pairs, but they are false positives. We also do not want to choose smaller distances because we still detect more true positives as we move closer to 0.15 threshold. This result is consistent with an earlier measurement reported in Figure 4.4.

We want to emphasize that the Jaccard distance threshold provides a desirable way to balance between efficiency and accuracy. Based on the resources available to scrutinize candidate piggybacked apps, we can adjust the distance accordingly. For a smaller dataset with less than 10,000 apps, we might choose to use a larger distance so as to catch as many piggybacked apps as possible. For a larger dataset with hundreds of thousands of apps, we might want to use a smaller distance to accurately return a high-density set that contains true piggybacked apps. In our above series of experiments, when we use the distance 0.15, our system reports 41 candidate pairs within 630 seconds and the distance of 0.05 returns 28 within 227 seconds.

### 4.3.4 Piggybacking Detection

From the previous section, we have empirically chosen 0.15 as the optimal Jaccard distance threshold. In this section, we apply it to our dataset and present our detection results. Overall, our system detects 1,094 (1.3%) piggybacked apps in our dataset. For these repackaged apps, we further obtain the corresponding carrier apps and then classify them based on their sources. The results are shown in Table 4.3.

In the table, the second column shows the number of piggybacked apps in each market and the third column contains the ratio of piggybacked apps to the number of apps we collected from each market.

<table>
<thead>
<tr>
<th>Jaccard Distance</th>
<th>0.05</th>
<th>0.1</th>
<th>0.15</th>
<th>0.2</th>
<th>0.25</th>
<th>0.3</th>
</tr>
</thead>
<tbody>
<tr>
<td>True Positives</td>
<td>28</td>
<td>38</td>
<td>39</td>
<td>39</td>
<td>39</td>
<td>39</td>
</tr>
<tr>
<td>False Positives</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

| Table 4.2: Determining the Right Jaccard Distance |
Table 4.3: Piggybacking Detection Results

<table>
<thead>
<tr>
<th>App market</th>
<th># Piggybacked Apps</th>
<th>Piggyback Rate</th>
<th># Carrier Apps</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slideme (US1)</td>
<td>49</td>
<td>1.6%</td>
<td>32</td>
</tr>
<tr>
<td>Freewarelovers (US2)</td>
<td>31</td>
<td>0.97%</td>
<td>52</td>
</tr>
<tr>
<td>Eoemarket (CN1)</td>
<td>224</td>
<td>2.7%</td>
<td>98</td>
</tr>
<tr>
<td>Goapk (CN2)</td>
<td>83</td>
<td>1.9%</td>
<td>108</td>
</tr>
<tr>
<td>Softportal (EE2)</td>
<td>39</td>
<td>1.7%</td>
<td>26</td>
</tr>
<tr>
<td>Proandroid (EE1)</td>
<td>32</td>
<td>1.9%</td>
<td>15</td>
</tr>
<tr>
<td>Official Android Market</td>
<td>683</td>
<td>1.0%</td>
<td>298</td>
</tr>
</tbody>
</table>

Due to the large number of apps it hosts, the official Android Market contains the largest number of piggybacked apps, but its piggyback rate is one of the lowest. The fourth column reports the number of carrier apps that have been chosen for piggybacking, which in general reflects which market is of interest to piggybacking authors in order to find popular apps to piggyback on. Our results show that game, wallpaper, and electronic book apps are among the most popular targets. Notice that the numbers in the carrier apps column are smaller than those in the piggybacked apps column. The reason is that the same carrier apps may be piggybacked multiple times to include different rider code (one concrete example is shown in the next section).

Interestingly, when examining these piggybacked apps inside the official Android Market, we find that 513 out of 683 (75%) are actually based on carrier apps also located in the official market. This clearly indicates the need for the official Android Market to adopt a rigorous policing to detect and potentially remove them. Also, notice that the remaining 170 piggyback on apps from third-party markets. This may sound counter-intuitive at first glance, but it is actually reasonable for two reasons: first, the official Android Market may not always be accessible or convenient to users outside the US, which partially explains the popularity of third-party markets in China; second, by choosing popular apps in third-party markets and uploading piggybacked apps into the official one, the app repackagers could reach more users for download and thus potentially maximize their impact.

To further measure the false negative rate, we study a list of 77 apps that were known to be piggybacked in our dataset (before our system was designed). PiggyApp correctly identifies 73 of them and misses four, indicating a false negative rate of 5.2%. Our manual analysis shows that these four failing cases are due to our module decoupling implementation, which incorrectly labels certain non-primary modules as primary and thus results in unnecessarily large Jaccard distances of related pairs.

4.3.5 Rider Analysis

After detecting these piggybacked apps, it is also interesting to find out answers to the following questions: what are the purposes behind these piggybacked apps? What rider code is injected into carrier apps? Are there (additional) permissions the rider code asks for? If there are, what are they? To answer
these questions, we perform a further analysis on these piggybacked apps. As it is not feasible to exam-ine every single piggybacked app, we choose to use cluster analysis to group and correlate the rider code.

The clustering analysis is motivated from our detailed investigation of these piggybacked apps. Specifically, when analyzing specific samples, we observe two common characteristics: first, many piggybacked apps share similar or even the same rider code; second, the same carrier apps are found piggybacked with different rider code. Our clustering analysis helps identify both of them.

In particular, to identify these common carrier apps, we simply count the number of each carrier app that occurs in the set of identified app pairs. One such example is a popular game app named `com.appspot.swisscodemonkey.steam`, which has been piggybacked on at least six times: four of them are variants of the `Pjapps` malicious payload [88], one is the `ADRD` malicious payload [9], and the other is an ad library named `wooboo` [90].

To locate the related rider code, we again apply our feature fingerprint technique to fetch the feature vectors of the rider code (there are 2067 of them present in 1094 different piggybacked apps) and apply the same nearest neighbor search algorithm (Algorithm 4). In this case, instead of choosing the previous threshold 0.15, we select 0.2 to loosely group rider code. As a result, we identify 16 clusters (ranging in size from 5 to 724). For each cluster, we randomly choose some samples for manual investigation. By doing so, we significantly reduce the time and effort needed to analyze them. From the analysis, it becomes evident the inclusion of rider code mainly serves two purposes. The first one is to inject various ad libraries with the intention to collect ad revenue or steal it from the original developers. The second one is to enclose malicious payloads to directly control compromised phones or steal personal information on the phones. In the next two sections, we examine these two purposes in more detail.

**Collecting Ad Revenue**

In the first purpose, the piggybacked apps are used to insert additional ad libraries, which help the repackagers, instead of the original developers, to collect ad revenues (generated from users’ views or clicks). As most of existing apps are free, developers want to monetize by including ad libraries and there are a variety of them [70, 84, 90], which are provided as standalone packages for simple reuse. Many of them require little or no change on the original code. Examples include `admob` [70] and `mobclix` [84]. Such convenience also makes it easy for repackagers to integrate them into popular apps as their carriers. Among the detected 1,094 piggybacked apps, 1,068 (97.6%) fall in this category. In Table 4.4, we show 13 top ad libraries in the rider code.

In the table, the first column shows the library name, the second column contains its detailed module name, while the third column counts the number of piggybacked apps that have it embedded. Among these 13 ad libraries, `admob` tops the list by being present in 724 carrier apps in our dataset. These ad libraries naturally request their own permissions for the provided functionality, some of which may not
Table 4.4: The Statistics of Piggybacked Ad Libraries

<table>
<thead>
<tr>
<th>Ad Library</th>
<th>Module Name</th>
<th># Piggybacked Apps</th>
</tr>
</thead>
<tbody>
<tr>
<td>admob</td>
<td>com.admob.android.ads</td>
<td>724</td>
</tr>
<tr>
<td>wooboo</td>
<td>com.wooboo.adlib.android</td>
<td>321</td>
</tr>
<tr>
<td>youmi</td>
<td>net.youmi.android</td>
<td>197</td>
</tr>
<tr>
<td>adwhirl</td>
<td>com.adwhirl</td>
<td>173</td>
</tr>
<tr>
<td>google/ads</td>
<td>com.google.ads</td>
<td>170</td>
</tr>
<tr>
<td>zestadz</td>
<td>com.zestadz.android</td>
<td>101</td>
</tr>
<tr>
<td>millennialmedia</td>
<td>com.millennialmedia.android</td>
<td>97</td>
</tr>
<tr>
<td>urbanairship</td>
<td>com.urbanairship.push</td>
<td>85</td>
</tr>
<tr>
<td>mobclix</td>
<td>com.mobclix.android.sdk</td>
<td>45</td>
</tr>
<tr>
<td>wiyun</td>
<td>com.wiyun.ad</td>
<td>36</td>
</tr>
<tr>
<td>mobclick</td>
<td>com.mobclick.android</td>
<td>26</td>
</tr>
<tr>
<td>greystripe</td>
<td>com.greystripe.android.sdk</td>
<td>26</td>
</tr>
<tr>
<td>madhouse</td>
<td>com.madhouse.android.ads</td>
<td>5</td>
</tr>
</tbody>
</table>

Table 4.5: The Statistics of Piggybacked Malicious Payloads

<table>
<thead>
<tr>
<th>Malware Family</th>
<th>Module Name</th>
<th># Piggybacked Apps</th>
</tr>
</thead>
<tbody>
<tr>
<td>Geinimi</td>
<td>com.geinimi</td>
<td>6</td>
</tr>
<tr>
<td>ADRD</td>
<td>com.xxx.yyy</td>
<td>1</td>
</tr>
<tr>
<td>Pjapps</td>
<td>com.android.main</td>
<td>8</td>
</tr>
<tr>
<td>DDream</td>
<td>com.android.root</td>
<td>10</td>
</tr>
<tr>
<td>BgServ</td>
<td>com.mms.bg</td>
<td>1</td>
</tr>
</tbody>
</table>

be requested by the carrier apps. It turns out that all these ad libraries ask for the INTERNET permission, 9 of them request the LOCATION permission, 5 need READ_PHONE_STATE, 1 demands CALL_PHONE, 1 uses ACCESS_WIFI_STATE, and 1 makes use of ACCESS_NETWORK_STATE. On average, these modules ask for 2.3 permissions.

**Injecting Malicious Payloads**

In the second purpose, repackagers implant malicious payloads into chosen carrier apps. In our dataset, we discover 5 different malicious rider payloads embedded in 26 different carrier apps that are present in various app markets. These malware are all listed in the yearly report of Android malware [121]. In Table 4.5, we show the list of detected malicious rider payloads. In the following, we choose representative samples and present our analysis.

Geinimi [82] is one of the earliest Android malware discovered in the wild that piggybacks on legitimate apps to perform malicious activities on the background. Our system identifies 6 piggybacked apps that have similar Geinimi code embedded (two different variants with 96% of their code in com-
mon). Both variants add their own activity, which once triggered invokes the embedded malicious code, including the bootstrap of a new service. These variants also add a new receiver to register for callbacks when certain events such as SMS_RECEIVED and BOOT_COMPLETED happen. By doing so, the malware can immediately run once the system boots or when a short message is received. To accomplish all these tasks, Geinimi needs 17 different permissions.

Compared to Geinimi, ADRD [9] is less complicated. Based on our analysis, the rider code is composed of four new receivers, which listen on the system boot completion event BOOT_COMPLETED, phone state change PHONE_STATE, network connection state change CONNECTIVITY_CHANGE and its own alarm timer com.lz.myservicestart. It also defines a new service that will send device-specific information to a remote server and receive instructions from it. In the piggybacked app, the main entry point remains the same as in the carrier apps. In our dataset, we only find one ADRD-piggybacked app with a new module named com.xxx.yyy. Overall, ADRD demands 7 different permissions.

Pjapps [88] is another malicious rider embedded in a number of carrier apps. In our dataset, there are 8 of them. All the related rider code share the same class package named com.android.main. In essence, it adds two new receivers and one more service. The internal mechanism of these new components works similar to that of ADRD. In total, Pjapps requests 9 permissions.

DroidDream [83] was reported in the official Android Market and our dataset contains 10 infected apps. Similarly, all of its rider code share a common module named com.android.root. As with Geinimi, it adds its own activity and starts a new service to set up an alarm timer, which in return triggers another service to launch one root exploit to elevate its privilege. With the root exploit, DroidDream requests fewer permissions, but can essentially do whatever it wants on the compromised devices.

The last piece of malicious rider code is from the BgServ malware, which injects one module named com.mms.bg into carrier apps to transport device-specific information (via short messages) to a remote party. One interesting thing about this payload is that it leverages an open source project hosted at Google code projects [1]. For its wrongdoings, BgServ asks for 9 permissions.

Overall, these malicious payloads all request more permissions than original carrier apps, which imply that the request for a bulk of permissions may be an indicator for potentially suspicious apps.

4.3.6 Performance

In this section, we report the performance measurement of our system. Our test runs on a Ubuntu Server 10.04 Linux machine with an four-core Intel Xeon CPU (2.67HZ) and 8G memory. Our current prototype runs on a single thread, which leaves room for future improvement to take advantage of multiple threads for speed-up.

In our test, we run the module decoupling and feature extraction separately, which is easily parallelized. Each app, depending on its complexity, takes from 0.167 to 5.492 seconds to process. On
average, it takes 0.952 seconds to process one app. Our module decoupling task of all these 84,767 apps takes 5 hours and 36 minutes in total. The construction of the VPT tree requires 126 seconds. Based on the VPT tree, given a single app, our algorithm takes between 0.00001 seconds and 0.576 seconds to find its nearest neighbor with 0.133 seconds on average. To iterate the apps in our dataset to locate possible piggybacked apps, it takes 3 hours and 15 minutes in total. The memory footprint seems small as only 127M main memory is used.

When streamlining the processing of these components, it takes less than nine hours to analyze our dataset with 84,767 apps from seven different Android markets. As mentioned earlier, improvements exist to better parallelize various components so that we can further reduce the processing time.

4.4 Discussion

Our prototype demonstrates promising results by allowing for fast and scalable detection of piggybacked apps. In this section, we examine possible limitations in the current prototype and discuss future improvements.

First, to infer piggybacking relationship, we extract semantic features based on primary modules of existing apps. These semantic features are mainly based on the Android APIs, requested permissions, and various intents, etc. Though these features satisfy our current needs, they are still limited in a number of ways. To improve the system, we could extend these semantic features to include syntactic instruction sequences [155] or control-flow graphs. The addition will be helpful to better characterize and identify a particular app. Meanwhile, our core prototype remains intact as we can easily expand feature vectors to accommodate them and reuse the same VPT tree for construction and lookup.

Second, our current prototype largely depends on the existence of authentic carrier apps in order to detect piggybacked apps. Unfortunately, due to the variety, scale, and dynamic nature of existing app markets, it is not possible to build a centralized repository having every app in existence; our current collection is far from complete. For example, there are cases where we can infer a potential piggybacking relationship but can not determine which one is actually piggybacked (Section 5.3). Also, our collection is comprised of only free apps and does not include paid apps, which are sold for their features and will likely be attractive targets for piggybacking. This also indicates the need to continuously expand our current data set with more comprehensive samples.

Finally, our current prototype basically serializes the execution of different components. As mentioned earlier, for improved performance, there is a need to re-engineer our prototype for a parallel version. Fortunately, the overall system design of PiggyApp is parallelizable in nature and does not require complete revamp.
4.5 Summary

In this chapter, we present PiggyApp, a system for fast and scalable detection of piggybacked apps in existing Android markets. Based on the observation that in a piggybacked app, the added rider code is loosely coupled with the primary functionality (or module) of the original app, we develop a module decoupling technique to effectively locate the primary module for comparison. To avoid pair-wise comparison, we further propose a scalable approach to extract semantic features from the decoupled primary modules and organize them in a metric space, which allows for fast and efficient search (with $O(n \log n)$ complexity). We have implemented a prototype and used it to detect piggybacked apps in a dataset collected from seven different markets. Our results show that 0.97% to 2.7% of apps hosted in these markets, including the official Android Market, are piggybacked. Based on these results, we further analyze rider code and find that it mainly serves two purposes: stealing ad revenue and implanting malicious payloads. These results call for a rigorous vetting process for their detection.
Chapter 5

AppInk: Watermarking Android Apps for Repackaging Deterrence

We have so far presented two systematic studies on understanding the app repackaging threats. Starting from this chapter, we present work that explores two different approaches to defend against app repackaging. This chapter presents a method to embed a watermark into the app’s running state to represent the author’s ownership. The next chapter (Chapter 6) presents a diversified virtualization-based protection for Android to thwart app repackaging.

5.1 Introduction

Facing the prevalent risks brought by app repackaging, we are in desperate need of a reliable, efficient, and easy-to-use mechanism to detect repackaged apps and prevent their propagation. Android package obfuscation tools such as Proguard [100] and DexGuard [86] have been provided by Google and other companies to confuse attackers when they are in the process of repackaging an app. However, obfuscation can only increase the difficulty of reverse engineering Android apps, and cannot stop determined attackers from achieving their purposes through manual analysis, laborious experiments, and strong persistence. To defend against Android app piracy and repackaging, Google has introduced a tool library named license verification library to protect app developers from having their apps stolen by third parties [71]. Other app store operators (such as Amazon and Verizon) also provide their own digital right management (DRM) options for app developers. DRM can be applied to prevent apps from being copied and pirated [122]. However, these mechanisms are hard to deploy correctly [49] and easy to crack [103–105]. Recently, academic researchers have introduced variant techniques to detect repackaged apps on a large scale [31, 60, 123, 154, 155]. However, these mechanisms usually cannot detect app repackaging online and in realtime. For these reasons, repackaged apps have usually been widely
distributed before being detected.

To overcome the weaknesses discussed above, we propose to embed a software watermark dynamically into the running state of an Android app to represent the author or developer’s ownership. For verification, an authorized party can extract the embedded watermark by running the app with a specific input in a dedicated environment, e.g., a customized emulator. When the extracted watermark matches the one provided by the developer, the verifying party (e.g., an arbitrator) can confirm the ownership of the original developer even when the app is repackaged by another publisher. The proposed watermarking mechanism should be resistant to manipulation by common static and automated attacks, thus making it hard for an attacker to remove the original embedded watermark or embed his own watermark.

To make this solution easily acceptable to the current app-centric system, we need two requirements: the watermark embedding should be easily integrated into current app development practices, and watermark extraction should be convenient for an authorized verifying party to perform. For the purpose of fulfilling these two requirements, we introduce the concept of manifest app, which is a companion app to the original app under protection. Basically a manifest app encapsulates a specific input to drive the watermark-protected app automatically, and thus eliminates the user interventions needed in traditional watermark extraction. Centering around the concept of manifest app, we design and implement a practical tool named AppInk, which consists of four components: watermarking code generation, automatic manifest app generation, watermark embedding, and watermark recognition. By seamlessly integrating these four components, AppInk presents an effective app protection solution for both developers and other authorized verifying parties.

Two usage models for AppInk  AppInk is designed to work in two usage models. The first is a central authority model. The second is post-mortem arbitrator model. In both models, the AppInk’s watermark embedder will be distributed to each app developer who wants to protect her apps. To do that, a developer needs to apply the first three components of AppInk to her app project. This generates two apps: the watermark-protected app to be submitted to centralized app markets or to be released through other channels to the public; and the manifest app, which is presented on demand to an authorized verifying party to verify the originality of the watermark-protected app. With the help of the watermark recognizer (the fourth component of AppInk), watermark verification will take place at different times in these two usage models.

In the central authority model, a centralized party (e.g., the app market) will review each submitted app to verify its originality before accepting it for publication. In this model, the watermark recognizer is pre-deployed at the app market’s site. Each app developer needs to submit the manifest app along with the app under review, and tell the app market the watermark value he embedded into the released app. The app market then runs AppInk recognizer, using the manifest app provided by the developer to drive the app under review. If the extracted watermark is the same as the one provided by the developer, app market will believe it is original and accept it for publication.

In the post-mortem arbitrator model, a third-party arbitrator will inspect the evidence of app owner-
ship to resolve dispute upon request. This usually takes place when one app author suspects that another app is a repackaged version of her own. In this case, the app author submits her manifest app to the third-party arbitrator, and requests to run the suspected app inside AppInk recognizer, using her own manifest app to feed input. If the watermark extracted from the suspect app is the same as the watermark extracted from the plaintiff’s app, the arbitrator is certain that the suspected app is repackaged from the app of the plaintiff developer.

To demonstrate the effectiveness of AppInk in deterring app repackaging, we analyze its robustness against general watermark-targeted attacks, including distortive, subtractive, and additive attacks. We also study its resistance against two open source transforming tools, i.e., ADAM [152] for repackaging purpose and Proguard [100] for obfuscation purpose. Our results show that AppInk is effective in defending against common automatic repackaging attacks. Our performance evaluation indicates that an embedded watermark introduces only a small overhead for end users.

5.2 Overview

5.2.1 Problem Statement

App repackaging refers to disassembling one app, making some modification to the code, data, or simply the signing key inside the original app, and rebuilding the modified components into a new app. As a technique method, it can be used for benign purposes. For example, ADAM [152] uses app repackaging to tweak malware samples for the purpose of stress testing various Android anti-virus tools. And Aurasium [148] uses app repackaging to intercept an app’s interaction with its underlying OS, aiming to enforce user-specified security policies for the app. However, app repackaging is more commonly used for surreptitious and malicious purposes. For example, mercenary publishers use app repackaging to replace existing in-app advertisements or embed new ones to steal advertisement revenues [31, 154, 155]. Malicious attackers use app repackaging [123, 155, 156] to plant malicious backdoors or payloads into benign apps to conduct malicious behaviors.

Because of the relative ease of reverse engineering Android apps (which are mainly written in Java), app repackaging has been identified as a widespread practice in current diversified app distribution channels [31, 60, 81, 123, 155]. As a result, it not only brings a lot of damage to app authors (e.g., monetary income and intellectual property), but also causes tremendous fear in the large community of mobile users and the burgeoning innovative app economy. As concrete examples, severe vulnerabilities have been found in mobile banking apps through app repackaging, and serious doubt is cast on mobile banking security and feasibility in general [96]. More recently, academic researchers have identified that about 10% of Android apps available in popular third-party markets are repackaged [155]. The latest investigation from industry [141] has reported that most of the popular mobile apps are beset by the app repackaging threat: 92 of top 100 paid apps for Apple iOS, and all the top 100 paid apps for Android

46
were found to be hacked.

Facing the widespread propagation of the app repackaging threat, effective security defenses are lagging far behind. The current industrial practices are either too weak to deter determined attackers from conducting repackaging attacks, or too complex to be deployed right and prone to be cracked. For example, obfuscation is recommended to app developers to protect their apps, but the introduced confusion usually is not strong enough to prevent firm attackers from achieving their goals [100]. App store operators (Google and others) have provided license verification or DRM service to apps submitted to their stores, but automatic repackaging tools can work around them easily [103–105]. Recently, researchers have begun to tackle this problem, but most of proposed solutions so far focus on feasible mechanisms to detect repackaged apps after their propagation [31, 60, 123, 155]. Considering the wide and severe impact of app repackaging, an effective and robust mechanism is intensely demanded to efficiently prevent and deter app repackaging in the first place.

5.2.2 Software Watermarking

Software watermarking has obtained extensive study to defend against piracy of desktop software [26, 28, 110, 113, 119]. With the similar process between app repackaging (in the mobile age) and desktop software piracy (in the Internet age), we believe software watermarking can be a promising technique in deterring app repackaging. Typically, watermarking software involves two steps: first, a watermark, which is some data (e.g., a number or a message string) known only to the author or publisher, is embedded into the target software in a specific way such that it does not affect the running behaviors of the original app and is difficult to remove without modifying the original app semantic; then, a special technique is used to extract the original watermark from the software. The matching watermarks verify that the software package belongs to the original developer or publisher. Depending on how the watermark is embedded and extracted, there are static and dynamic watermarking techniques. Static watermarking embeds a watermark into the code or data of a package, and extracts the watermark without executing the code. Dynamic watermarking embeds a watermark into the execution state of the target software, and extracts the watermark during runtime.

Regardless of which method is used, it is desirable that a watermark embedded in a software package cannot be easily compromised by well-known attack techniques, including distortive attacks – to apply semantic preserving transformation on the watermarked code to modify the embedded watermark; subtractive attacks – to remove complete or partial watermark; and additive attacks – to add attacker’s own watermark and confuse the arbitrator on resolving ownership dispute. Dynamic watermarking usually represents the watermark as a special graph structure on the runtime heap, and exploits the undecidability of pointer analysis to enhance its robustness against different attacks. For that reason, dynamic watermarking in general has a stronger resistance against these attacks than static watermarking, and it is used in our approach to embed and verify the ownership of a specific Android app package.
Pointer Analysis As a program analysis technique, pointer analysis attempts to statically determine the possible runtime values (or storage locations) of a pointer (or a heap reference). Since pointers (or references) are used a lot in program structures, pointer analysis becomes a prerequisite for performing many of the common program analysis tasks (such as reaching definition and live variable analysis). For programming languages with if statements, loops, dynamic storage and recursive data structures (such as C and Java), computer scientists have established that pointer analysis is undecidable [101, 128]. However, for specific usages there are also a lot of approximate algorithms that have good tradeoff between efficiency and precision. For example, some flow-insensitive or context-insensitive pointer analyses [24, 62, 131] are very efficient, practical enough for the usage of some client analyses (such as program optimization and error detection). However, the precision of these efficient algorithms has been a problem [61, 62], making them impractical for the usage of other client analyses. To attack AppInk, precise pointer analysis is in need to achieve a fruitful result.

In AppInk, we encode the watermark as some code segments (cf. Section 5.3.2) to represent a permutation graph, which is composed of a series of dynamic allocated objects (nodes) and dynamically generated object references relationships (edges) on the runtime heap. As in common object-oriented programs, object reference is ubiquitous in Android apps. Many Android objects have explicit references to other objects, and the Android framework also maintains a lot of object references implicitly to support the routine work of these running apps. Also, there are a series of Android classes that maintain references to objects of their own types. For example, java.util.Arrays, java.util.LinkedList, java.util.HashSet, and many other collection classes are used in common Android apps to organize objects having similar features. Overall, ubiquitous object references in the common Android app makes the embedded watermarking code hard to distinguish from the original code, and thus hard to analyze and remove.

To identify, modify or remove these watermarking codes, attackers usually have to conduct precise pointer analysis to understand how the app is working and how different pieces of code are interacting with each other. In general, these analyses have to be bound to specific points at specific execution paths so that attackers can understand which pieces of code are more tightly coupled with others. In this sense, the flow-insensitive pointer analysis that does not consider control flow, and context-insensitive analysis that does not consider calling context, will not help the attackers in separating these stealthy watermarking codes from the original code. However, precise pointer analysis is proved to be undecidable [101, 128] for programming languages like C and Java (These are also the current languages used by Android developers.).

What is more, AppInk can use other tricks to make it harder for attackers to conduct the pointer analysis and to separate the watermarking code. For example, we can introduce a large number of artificial object references through inserting reference-intensive code. We can use opaque predicates [29] (which can be generated similarly based on the difficulty of precise pointer analysis) to tightly couple the control flow of the original code with the watermarking code segments. For a more detailed analysis
of AppInk’s robustness against common attacks, please refer to Section 5.5.1.

5.2.3 Challenges of Watermarking Android Apps

There are several key challenges to incorporate dynamic watermarking into current Android app development practice and make it easily deployable by arbitrating parties.

Firstly, the state of the art dynamic Java watermarking techniques need extensive intervention from developers to embed a watermark. For example, SandMark [27] needs the developer to manually annotate source code to indicate where the watermark can be inserted, and manually give input to drive the software when embedding a watermark. These manual interventions make it cumbersome to apply this technique correctly in real practice.

Secondly, it is desirable to have automatic watermark recognition so that it can handle thousands of apps with little human effort in an online and realtime manner. To recover embedded watermarks, SandMark leverages programmable Java Debug Interface [85] to access memory objects on the heap in order to infer object reference relationships. However, there is no known programmable debugging interface available on Android. Even worse, manually giving input is required for the watermark extraction phase as well. Obviously, this cannot scale to handle watermark recognition for thousands of Android apps submitted to the current app store everyday.

Thirdly, Android apps, although mainly developed with Java language, have significant differences from desktop Java software. For example, Android apps depend more heavily on event-driven mechanisms and the underlying execution environment to work correctly. So we cannot simply reuse watermarking techniques created for desktop software. Also, unlike legacy Java applications that have a single entry point named `main`, Android apps in general have multiple entry points.

5.2.4 Solution Overview

To overcome these challenges, we introduce an entity named `manifest app`, which is a companioning app for a target app under protection. Basically a manifest app encapsulates a sequence of input events to drive the watermark-protected app automatically, and thus eliminates the user intervention needed in traditional watermark extraction. Based on the manifest app approach, we design and implement a practical tool named AppInk to automatically generate the manifest app, embed the watermark, and execute the dynamic watermark extraction with zero user intervention. As an input during watermark embedding, the manifest app encodes the event sequences and accordingly indicates the event handlers of the target app where watermarking code segments can be inserted. This resolves the first challenge. As an input for watermark extraction, the manifest app automatically launches the original app and feeds the input event sequences to it, which triggers all the inserted watermarking code segments and thus recovers the watermark object embedded. This resolves the second challenge. Based on the insight that each event in the Android platform is uniquely mapped to a well-known system API, we propose a conservative
method to automatically generate an event flow model for Android apps, and leverage model-based test generation to automatically create a suboptimal manifest app for watermarking purpose. This resolves the third challenge.

5.2.5 Trust Model

In the framework of AppInk, manifest app is not released to the public. We assume that it is kept secret by relevant parties. After generating the manifest app for a specific app, the app developer only sends it to arbitrating party on verification request. The parties who receive the manifest apps are either the app market (in the central authority model) or the arbitrator (in the post-mortem arbitrator model). Because of their unique roles, AppInk assumes that they are trusted, will not tell lies about the app’s authorship, and also will not send the app developer’s manifest app to other parties. The app developers who have submitted their apps to the app market are not trusted. They might download and repackaging other developers’ apps and resubmit them. They have access to AppInk’s watermark embedder (in binary) and might be able to reverse engineer it. Normal mobile users are also not trusted. They can download, install and play with the released app protected by AppInk. They can also apply different analysis tools to the apps, and even attempt to add their own watermark into the app.

5.3 Design of AppInk

This section first gives an overview of AppInk architecture with its major functioning components, followed by their design details.

![Figure 5.1: The Overall AppInk Architecture.](image-url)
5.3.1 Architecture

Figure 5.1 depicts the overall AppInk architecture, centering around the manifest app. At the app developer (left) side, AppInk consists of three components: manifest app generation, watermark code generation, and watermark embedding based on source code instrumentation. The input of the manifest app generation component is the source code of the target app, including its resource files. The watermark code generation takes a watermark object (e.g., a number or a string) and outputs watermarking code segments. The watermark embedding component takes the manifest app and the code segments as inputs, and generates a watermarked Android package that can be released to app stores. At the arbitrator (right) side, the watermark recognizer takes the inputs of the released Android package and the manifest app from the app developer side, and extracts watermark.

**Watermarking code generation:** Given a watermark value specified by the app developer, this component encodes the watermark value into a special graph structure and transforms the graph into watermarking code. In order to improve the stealthiness of the embedded watermark, AppInk splits the watermarking code into a variety of segments, each of which will be inserted into different locations of the original app’s source code. The execution states of this code collaboratively present specific object reference relationships and thus can be leveraged to reveal the original watermark value. Section 5.3.2 explains in detail the design of this component.

**Manifest app generation:** The main function of the manifest app is to feed pre-determined user inputs to the app under review, which trigger the executions of embedded code segments and thus recover the watermark value with the help of the watermark recognizer. To ease the burden of writing manifest apps by developers, AppInk leverages the event-driven nature of Android apps and the latest model-based test case generation to automatically generate these input events and makes this process totally transparent to app developers. Section 5.3.3 presents in detail the design of this component.

**Source code instrumentation:** Through parsing the files in the manifest app, this component first identifies encoded user input events, and then determines their corresponding event handlers through analyzing the source code of the original app. It then insert the watermark code segments under the path of these identified event handlers. After this step, AppInk packages the modified app source into a released app, which is for both public publication and arbitrating purposes, and the manifest app into another executable package, which is not released to the public but dedicated to arbitrator consumption. Section 5.3.4 presents in detail the design of this process.

**Watermark recognizer:** This component is a modified Android emulator on x86 [75], which is invoked by a script. The script first installs both the Android app and the manifest app in the emulator, then starts the manifest app which feeds a sequence of input events to the Android app, and then calls the extended Dalvik Virtual Machine (DVM) [34] to export all object reference information in the runtime heap. From this information, AppInk uses a special pattern to match potential watermarking structure. If such a watermarking structure is identified, a reversed process of the watermarking code generation is invoked.
to recover the embedded watermark value. Section 5.3.5 illustrates the design of this component.

### 5.3.2 Watermarking Code Generation

Different from static watermarking, which embeds a secret watermark object (e.g., a numerical value or a message string) into the code or data section of a target application, dynamic watermarking embeds a watermark object into special structures that present themselves only in the runtime of the target application. AppInk adopts a graph-based data structure, which is hard for attacker to reverse due to the inherent difficulty of precisely analyzing point-to-relationships in graphs [50, 128].

There are different ways to use graph to encode a watermark object. AppInk uses a permutation graph, which adopts a special graph structure to encode a permutation mapping to the watermark object. As depicted in Figure 5.2, the graph includes 5 nodes, each of which has two outgoing edges, one in a solid line and one in a dotted line. Through the solid line edges, the graph forms a cycle. If we can further identify one unique node (e.g., the one which is referenced by any other object outside the figure), a specific order is defined. Supposing that only the first node is referenced by another object not in the figure, we can assign number 0 to this node, which is called root node, and numbers 1 to 4 for the other four along the circle. A dotted line edge of a node is then associated with a number counting the distance from this node to its target node along the solid line edges. For example, the dotted outgoing edge from node0 to node1 encodes a number of 1 since the distance from node0 to node1 is 1 along the solid line edges. Similarly, the dotted outgoing edge from node3 to node1 encodes a number of 3. In Figure 5.2, the 5 dotted edges encode the numbers 1, 2, 0, 3, and 4, respectively, which is a perfect form of permutation. According to the permutation-to-number algorithm in [97], <1, 2, 0, 3, 4> is mapped to 116, which is the watermark value encoded by this graph.

In the Java language, the permutation graph depicted in Figure 5.2 can be represented by a double linked list, as shown by the skeleton code in Figure 5.3. The class `WatermarkNode` (lines 1 to 5) represents the node in the permutation graph. The following initialization code (lines 7 to 9) creates five instance nodes, and the final code (lines 11 to 15) defines the object reference relationship, from which we can reconstruct the permutation graph in the runtime. As commented in the list, member field `solid` points to the next node in the list, and all these fields form a cycle. Member field `dotted` encodes the
class WatermarkNode {
    WatermarkNode solid;
    WatermarkNode dotted;
    .......
}

node0 = new WatermarkNode();
......
node4 = new WatermarkNode();

node0.solid = node1;   // Points to next node
node0.dotted = node1;  // Encode number 1
......
node4.solid = node0;
node4.dotted = node3;  // Encode number 4

Figure 5.3: Watermarking Code for the Permutation Graph in Figure 5.2.

permutation distance for each node in the permutation graph, which jointly encodes the watermark value specified by the app developer.

When the above code is executed on the Android platform, memory space will be allocated for each instance object (node0 to node4) at lines 7 to 9. At lines 11 to 15, the member fields are assigned, which result in the establishment of the object reference relationship among these WatermarkNode instances. Through analyzing the runtime heap, this object reference relationship can be extracted and decoded to recover the original watermark value. Section 5.3.5 presents the details of this process in the watermark recognizer.

Because linked structures are very commonly used in Java applications, it is hard to distinguish this watermarking code from other codes. The stealthy nature of these graph structures, combined with the inherent difficulty of precise pointer analysis in the graph [50,128], makes it very challenging for attackers to succeed in reverse engineering the watermarked code. A more detailed analysis of the robustness of this technique is presented in Section 5.5.1. To further improve the stealthiness of the watermarking code, AppInk splits the watermarking code into a number of segments and inserts them into a variety of places in the app. This is especially helpful when the watermark value is large and thus has to be represented by a large number of code segments. Section 5.3.4 presents more details of this technique with the help of manifest app.
5.3.3 Manifest App Generation

Manifest App Based on Robotium

Working as a companioning app to drive the execution of a released app inside a watermark recognizer, a manifest app functions in a similar way to test cases. However, unlike common unit tests, which only provide component-specific tests [78], and special Android UI/application exerciser [80], which sends a random stream of events to apps under test, a manifest app needs programmable event delivery within the entire target app, so that watermarking code can be scattered to different places and thus be more stealthy. For this purpose, AppInk generates a manifest app based on Robotium [129], which extends the Android app instrumentation framework and provides precise UI element locations and event delivery. Figure 5.4 shows an example Robotium test case.

In Figure 5.4, method setUp starts the main activity of the app under test, tearDown clears the execution environment and stops its execution, and testEventSequence (line 5 to 9) sends a specific sequence of events to the app. To automatically generate manifest apps and later extract the watermark by the recognizer, AppInk needs to decide a proper input sequence and fill them into method testEventSequence. Specifically, AppInk has two requirements for the input sequence: its execution can deterministically trigger the watermarking code segments, and it is diversified enough so that the watermarking code can be scattered into a large enough scope. AppInk leverages the event-driven nature of Android apps and model-based automatic test case generation to achieve these purposes.

Manifest App Generation

Different from desktop applications, the control flow of Android apps heavily depends on the diversified Android events, including user generated events (e.g., key press and screen tap) and system generated events (e.g., short message received, incoming phone call, and various sensor events). Each event is handled by a well-defined Android API. For example, a menu item click is handled by method
onOptionsItemSelected in the corresponding activity, a button press is handled by the onClick method of the listener object registered for the button, and a short message received event is handled by method onReceive of the activity registered for SMS_RECEIVED intent. By issuing these events in a well-defined order, the app under test invokes these event handlers in order, and responds in a deterministic manner.

For the purpose of automatic test generation, model-based methods have been well studied for event-driven software in general, and actively investigated for Android apps in particular [95, 109, 115, 138]. But to use it in AppInk, we need to have the app’s event flow model as input. One solution is to ask developers to provide the model. But this puts an extra burden on them, and is also prone to error. Another option is to infer the event flow model through reverse engineering. We note that generating a complete app model is a hard problem as studied by many researchers. However, unlike common test case generation, whose task is to exhaustively generate test sets to cover as many paths as possible, AppInk just needs one test case if it can trigger as large a set of code segments as possible to achieve stealthy watermark embedding. Therefore, we only need to have a partial model for the app event flow.

AppInk uses a static method to infer a partial event flow model for Android apps in a conservative but safe way. This is achieved through parsing app source files, including AndroidManifest.xml, UI layout, Java and other resource files. The generated model is fed to an existing model-based test generator for Android [45] to generate a set of test cases, from which we pick the test input covering the most code segments.

We now use an example app to describe how AppInk generates the event flow model. Figure 5.5 shows the user interface elements and relevant events for app NotePad. The first screen (5.5a) pops up right after the app starts. By analyzing the layout and Java source files, we infer that event Add note is handled by method onOptionsItemSelected in file NotesList.java. The second screen (5.5b) shows the UI elements for the action of adding a note, including a text input box and two menu items (Save and Discard), whose event handlers are method onOptionsItemSelected in file NoteEditor.java. The third screen (5.5c) shows the UI elements for editing a note, including a text input and three menu items (Save, Delete, and Edit Title), each of which has its own handler. The last screen (5.5d) shows the UI elements for editing title, including a text input and one button (Ok), whose handler is onClick method in file TitleEditor.java.

At this point, each screen shows only individual events. Through analyzing the handler for the event of clicking Add note (method onOptionsItemSelected in file NotesList.java), AppInk determines that it starts an activity with the intent of ACTION_INSERT, which is found later to be defined in file NoteEditor.java through parsing file AndroidManifest.xml. So event Add Note connects screens 5.5a and 5.5b. Through similar analysis, we determine that the event of clicking list item connects screens 5.5a and 5.5c, and the event of clicking Edit title connects screens 5.5c and 5.5d. Furthermore, there is a back button below the display screen of the phone, which connects the current screen and the one before it. Having these connecting events, AppInk generates the event flow
Figure 5.5: User Interface Elements in App NotePad.
Figure 5.6: Event Flow Graph for NotePad.

```java
public void testEventSequence() {
    clickOnMenuItem("Add note");
    enterText(0, "Test");
    clickOnMenuItem("Save");
    goBackToActivity("NotesList");
    clickInList(1);
    clickOnMenuItem("Edit Title");
    clickOnButton("Ok");
    clickInList(1);
    clickOnMenuItem("Delete");
}
```

Figure 5.7: Skeleton Code to Drive NotePad.

model as depicted in Figure 5.6.

After feeding the above event flow graph into M[agi]C [45] — a test input generator tool, we obtain a test case with the skeleton shown in Figure 5.7. This skeleton encodes the event sequence of Add note, Enter text, Save, Back, Click list item 1, Edit Title, Ok, Click list item 1, and Delete. This sequence covers all the activity classes in the app, thus presenting an optimal test for watermarking purposes.

### 5.3.4 Source Code Instrumentation

Having the manifest app source, together with the original app source and the generated watermarking code, AppInk uses source code instrumentation to perform the watermark embedding. The choice of source code instrumentation is reasonable since AppInk is used by app developers who already have the app source code at hand. This also helps integrate AppInk with the well-established app development environment for Android. With this said, please note that the AppInk framework is also well supported
by bytecode level instrumentation.

Source code instrumentation uses three steps to embed developer-provided watermarks. First, AppInk fetches all control events (including clicking button, menu and list items) from the manifest app, each of which is mapped to a single event handler in the original app (such as onOptionsItemSelected and onClick). Next, AppInk splits the watermarking code into the same number of code segments as the number of the event handlers, and generates a configuration file to record the one-to-one mapping from the watermarking code segments to the event handlers. Last, AppInk parses the source code of the original app, generates its abstract syntax tree, identifies nodes for the event handlers, and inserts the watermarking code segments into their corresponding event handlers.

After the above instrumentation, AppInk automatically builds and generates an executable app package and signs it [77], which can be used for public release. The manifest app is built into another executable package and not released into public. Instead, it will be submitted upon request to the arbitrator for verification purposes. All these steps are put into one script, which is seamlessly integrated into the app building process.

5.3.5 Watermark Recognizer

As depicted in Figure 5.8, the recognizer has both apps as input: a released app for reviewing and its manifest app as the driver. The core part of the recognizer is an extended Dalvik virtual machine (DVM), which is the execution engine for Android app code and maintains a runtime heap. Unlike the traditional watermarking tool for Java that uses the Java debug interface to access object reference information and reconstruct the watermark value, AppInk leverages the customized DVM to fetch object reference information from the runtime heap directly. With this and the help of the manifest app generated in Section 5.3.3, the watermark recognizer enables automatic watermark extraction without any user intervention, which is highly desirable for scalable handling of large number of apps submitted to app stores.

Just like Java virtual machine executing Java code by interpreting bytecode, DVM executes Dalvik bytecode, which is the main body of Android apps. Therefore, it has access to all of the needed information for watermark extraction purpose. Particularly, DVM manages memory space for Android apps, and maintains relevant information for memory reclaim (garbage collection). All object reference information is maintained there, so that AppInk only needs to extend the garbage collector to record and export this information, among which a later module will search for the watermarking graph. The identified watermarking graph is then decoded to recover its corresponding watermark object and to verify that it is the same as what the author claims.

More specifically, DVM uses a mark-and-sweep algorithm [147] to execute the task of garbage collection, which scans all allocated objects and their member fields (a reference relationship forms between an object and its member fields) and determines if any object is not needed any more and thus
can be reclaimed. We develop a module in DVM to record these object reference relationships in the scanning phase and export them into a log file. From these reference relationships, a reverse process to that of the watermarking code generation (cf. Section 5.3.2) is applied to recover the watermark value.

Towards automatic operations, we create a shell script based on Android debug bridge [73] to link all these steps, as shown in Figure 5.9. The script first installs both apps – the released app and the manifest app (lines 1 and 2), and then starts the manifest app through the instrumentation command of the Android activity manager (line 3). This will feed the event sequence to the released app in a specified order. The command at line 5 gets the process identifier of the running DVM, and line 7 sends a SIGUSR2 signal to trigger the object reference recording module inside the extended DVM. The commands at lines 8 and 9 fetch these recorded messages, search reference relationship pattern among them, and try to extract embedded watermark.

```bash
adb install -r releasedApp.apk
adb install -r appTest.apk
adb shell am instrument -w InstrumentTestRunner

pid='adb shell ps|grep appName| awk '{print \\2}'
## Send USR2 signal to trigger GC
adb shell kill -10 $pid
adb logcat -d | grep $pid > $pid.log
java appink.wmGraphRecognizer $pid.log
```
5.4 Implementation

We have implemented an AppInk prototype on Ubuntu. The watermarking code generation component is implemented in Java, which accepts a big integer or string as input and outputs its corresponding watermarking code. In the component of manifest app generation, the parsing of Java source files is based on ANTLR [120] – a language parser generator. More concretely, we input a Java language grammar [59] into ANTLR, which generates a Java AST (abstract syntax tree) parser. By iterating through the AST, AppInk can locate the nodes for all event handlers, and identify the connecting events for different UI states. The parsing of AndroidManifest.xml, UI layouts, and resource files is written in Python. Another Python script glues the output results from these parsing modules, generates the event flow graph, feeds it to the test case generator named M[agi]C, and picks up the test case that has the largest coverage of watermarking code.

The source code instrumentation component includes three steps. It first parses the manifest app source to identify all of these event handlers where to insert the watermarking code, and then splits the watermarking code into segments with the same number as that of the events to be delivered. Last it extends the Java source parser generated in the manifest app generation component to insert the watermarking code segments into the execution path for their corresponding event handlers. To automate the watermark extraction at arbitrating side, this component also generates the shell script to drive watermark extraction (as presented in Figure 5.9). Basically, this script only needs relevant information for a released app and its manifest app, which is readily available after the completion of the first three components.

Having this watermark extraction script at hand, the watermark recognizer needs two modules to achieve the final watermark recognition task. The first module implements the extended DVM to record and export object reference information when the app receives a SIGUSR2 signal. This is achieved by modifying the garbage collector code (in C language) in DVM and rebuilding the Android open source project. The second module is written in Java, which searches through these object reference relationships to match any potential watermarking graph, and decodes the graph to recover corresponding the watermark value.

5.5 Analysis and Evaluation

While AppInk aims to embed strong ownership verification mechanism into Android apps, attackers always strive to defeat the protections in any way they can think of. In this section, we first analyze the robustness of AppInk against three common attacks toward watermarking, namely distortive, subtractive, and additive attacks. We then evaluate it against two open source repackaging tools to demonstrate its effectiveness. Finally we evaluate the runtime performance overhead for watermarked apps.
5.5.1 Robustness Analysis

AppInk adopts dynamic graph based watermarking as its key technique to defend against app repackaging. Therefore its robustness heavily depends on that of dynamic graph watermarking, which is highly resistant against distortive attack, subtractive attack, and additive attack according to our analysis.

**Distortive attacks:** This type of attacks applies semantic-preserving transformations on target apps, trying to make it hard or impossible to extract the original watermarks from the modified apps. Many static watermarking mechanisms are highly susceptible to distortive attacks since they leverage the code or data syntax to encode a watermark, which is very sensitive to semantic-preserving transformation. Dynamic watermarking, however, never depends on any syntax structure in application code, but instead encodes a watermark object into the execution state of an application. Furthermore, the semantic of runtime graph data structures is usually hard to analyze without executing it in a real environment, because of the inherent difficulty in analyzing point-to relationships [50, 128]. These factors make it very hard for any static transformation to change these graph structures without changing the application semantics. For these reasons, most semantic-preserving transformations cannot affect the execution state of apps, and theoretically dynamic graph watermarking is resistant to distortive attacks. To further confirm AppInk’s robustness in this aspect, we evaluate AppInk against a series of semantic-preserving transformations available in two open source tools ¹, and report our results later in this section.

**Subtractive attacks:** This type of attacks tries to remove watermarking relevant code segments in an application, and usually needs manual analysis to identify the locations of these code segments in the first place. The dynamic graph based watermarking mechanism adopted in AppInk makes it relatively easier to defend against subtractive attacks. First, the data structures used in AppInk are commonly used in normal Java applications, which makes it hard to separate these watermarking code segments from other functional code. We can also leverage the inherent difficulty of alias analysis [50, 128] to add another layer of protection against subtractive attacks. Since the runtime graph data structures in AppInk have reference relationships among themselves, an app developer can easily know the correct reference relationships among the inserted graphic nodes. Instead, without this pre-knowledge, attackers have great difficulty to identify these reference relationships through reverse engineering. Therefore we can create bogus dependency relationships between the original code and the newly inserted code [29]. The attempt to remove or modify the watermarking graph code segments will have a high probability of damaging the original application logic, making the app useless after repackaging.

**Additive attacks:** This type of attack tries to add another watermark on a watermarked app, with the assumption that attackers somehow understand and implement the same watermark embedding algorithm as presented in AppInk. In the central authority model (cf. Section 5.1), there is an inherent timing gap between the submission of the original app and the repackaged app. That is, the original author always submits her AppInk-protected app before an attacker succeeds in executing additive attack on that app.

¹They can be used for app repackaging as well.
Figure 5.10: Watermark Embedding, App Repackaging & Watermark Recognizing Snapshots
Therefore when the attacker submits his repackaged app to the app store, the app store can detect that there is an earlier app which has the same functionality but has a different watermark extracted. The operator can then launch another watermark extracting session on the app under review, using the manifest app for the earlier app as the watermark extraction driver. If the same watermark is extracted as the earlier published app, it is derived that the second is a repackaged one. However, this does not prevent the attacker from downloading an app from one store and publishing in another one where the original app has not been published yet.

In the post-mortem arbitrator model, when an app author suspects that one app is a repackaged version of her own, she can apply the watermark recognizer with her own manifest app on the suspected app. If a same watermark is extracted as from her own app, she can submit this as evidence to prove that the suspected app is a repackaged version of her own. In case that an attacker can somehow embed his own watermark into the AppInk-protected app and generate his own manifest app, two various watermarks can be extracted through using their corresponding manifest apps. Under this confusion, the original author can present another evidence to show that her watermark is original, and the other is additional. The evidence is that her manifest app can extract the same watermark from both her original app and the repackaged app, but the attacker’s manifest app can only extract his own watermark from the repackaged app.

Next we evaluate the AppInk’s robustness against real repackaging tools. With our best effort we do not find any available tool for subtractive and additive attacks, therefore our evaluation is for distortive attacks only.

5.5.2 Evaluation with Repackaging Tools

We evaluate AppInk against two open-source tools, which apply semantic-preserving transformations on Android app code and therefore can simulate the aforementioned distortive attacks. We have five Android apps under evaluation, one named AndroidCalculator from Robotium, and the other four from Android SDK samples (including ContactManager, NotePad, HoneycombGallery, and SearchableDictionary). We first apply AppInk to embed watermarks on these apps, and then apply the available transformations present in the above tools to the watermarked apps. Last we feed the modified apps to the AppInk watermark recognizer to see if the originally embedded watermarks can be extracted.

ADAM: We first evaluate AppInk against an automatic Android app repackaging tool named ADAM [152], which operates on Android apps (.apk files) directly and automatically repackages apps with different code transformation techniques. Figure 5.10 shows the above three-step evaluation process for the app named NotePad. The first snapshot (5.10a) shows the embedding session on NotePad.apk, with a string of 1234567890abcdef as the watermark value. It clearly demonstrates the working process of the three AppInk components.
The second snapshot (5.10b) shows the repackaging session, where we apply seven semantic-preserving transformations from ADAM on NotePad.apk. Among these transformations, three do not modify the app code but change other phases in the app packaging process. For example, resign transformation disassembles the app and resigns the app using attacker’s signing key, rebuild transformation disassembles the app and re-assembles the components into a new one with the open-source tool named apktool [126], and zipalign transformation realigns the locations of different data in the app package in a different pre-determined way. Four other techniques apply various code obfuscation transformations on the app code, including inserting defunct code, identifier renaming (including packages, classes, methods, and fields), control flow obfuscation, and string encryption. To conduct the evaluation, we apply these seven transformations on the watermarked code with a bash script, and output the repackaged apps into a directory.

The third snapshot (5.10c) shows the recognizing session. We create a script to feed each of these seven repackaged apps into the watermark recognizer, and check the extracted watermark value. The first attempt shows that AppInk recognizes the watermark value of 1234567890abcdef correctly from six of these repackaged apps. The app repackaged with identifier renaming obfuscation fails the first attempt. Later analysis shows that ADAM incorrectly renames one Android API method, which results in the incorrect execution of the repackaged app. After fixing this bug in ADAM, AppInk recognizes the correct watermark from all these repackaged apps (Figure 5.10c shows the result after fixing the ADAM bug).

**Proguard:** Our second evaluation is against a popular Android app obfuscation tool named Proguard [100]. Different from ADAM which works directly on final .apk files, Proguard operates on class files generated in the Android app building process. To conduct this evaluation, we modify the watermarking embedding process as presented in Section 5.3, by adding Proguard obfuscation as a post-compilation action into the app building process. With this extra action, the generated class files are optimized and obfuscated first, and then packaged into the final released apps. Last, we feed these obfuscated apps into the AppInk recognizer. Our experiments show that AppInk recognizer can extract the correct watermarks embedded into all of these transformed apps successfully.

These two sets of evaluations demonstrate that AppInk has high resistance against currently available repackaging and transformation tools, and thus is very robust against distortive attacks.

### 5.5.3 Performance Evaluation

AppInk inserts some watermarking code into the apps under protection, which incurs extra overhead for the running app. The watermark extractor of AppInk also takes time to finish. In this section we present the experimental results to evaluate the performance overheads in these two aspects. All the experiments are conducted on an emulator running Android v2.3. Each experiment is performed 40

---

2Concretely, we add a new Proguard configuration file and Proguard action into one ant [77] building script.
Table 5.1: App Execution Times and Extra Delays (in Seconds)

<table>
<thead>
<tr>
<th>App Name</th>
<th>Original App</th>
<th>WM-5nodes (extra delay)</th>
<th>WM-15nodes (extra delay)</th>
<th>WM-25nodes (extra delay)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ContactManager</td>
<td>6.85 ± 0.21</td>
<td>6.89 ± 0.19 (0.6%)</td>
<td>7.01 ± 0.25 (2.3%)</td>
<td>7.03 ± 0.18 (2.6%)</td>
</tr>
<tr>
<td>SearchableDictionary</td>
<td>5.37 ± 0.12</td>
<td>5.39 ± 0.18 (0.4%)</td>
<td>5.45 ± 0.16 (1.5%)</td>
<td>5.53 ± 0.21 (3.0%)</td>
</tr>
<tr>
<td>NotePad</td>
<td>25.7 ± 1.2</td>
<td>25.9 ± 0.98 (0.8%)</td>
<td>25.9 ± 1.1 (0.8%)</td>
<td>26.3 ± 1.0 (2.3%)</td>
</tr>
</tbody>
</table>

Figure 5.11: Execution Time of Watermarked App.

times and average results with 95% confidence intervals are reported in the tables below. Three Android apps are used in our experiments: ContactManager, SearchableDictionary, and NotePad. As the length of the permutation graph is the main factor to decide the extra code size and thus the final performance, we watermark each of these three apps with five different watermark values, which encode permutation graphs with length of 5, 10, 15, 20, and 25, respectively. These values can encode a number from $24$ to $1.6 \times 10^{25}$. Our experiments show that even the longest watermark value only introduces trivial performance overhead.

First, we measure how much extra time is required to execute these watermarked apps, which affects the experience of an end mobile user when running these AppInk protected apps on their devices. To reduce the undecidability of human input and also exercise all of these watermarking codes, we use the manifest apps to drive these apps in a normal Android emulator. Table 5.1 shows the average overhead for the original apps and some of these watermarked apps. The extra delays introduced by AppInk are shown in the parentheses for each watermarked app. They range from 0.4% to 3.0%. Figure 5.11 shows a more intuitive comparison of these execution times. Please note that for each app, the first
Table 5.2: Extraction Times of Watermarked Apps (†: the number in parenthesis shows the difference between app execution time and watermark extraction time.)

<table>
<thead>
<tr>
<th>App Name</th>
<th>WM-5nodes (in seconds)</th>
<th>WM-15nodes (in seconds)</th>
<th>WM-25nodes (in seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ContactManager</td>
<td>8.23 ± 0.18 (1.34†)</td>
<td>8.41 ± 0.21 (1.40†)</td>
<td>8.47 ± 0.23 (1.44†)</td>
</tr>
<tr>
<td>SearchableDictionary</td>
<td>6.67 ± 0.20 (1.28†)</td>
<td>6.89 ± 0.23 (1.44†)</td>
<td>7.21 ± 0.21 (1.68†)</td>
</tr>
<tr>
<td>NotePad</td>
<td>27.1 ± 1.1 (1.20†)</td>
<td>27.3 ± 1.3 (1.40†)</td>
<td>27.6 ± 0.99 (1.30†)</td>
</tr>
</tbody>
</table>

column shows the time to execute the original (un-watermarked) app. The small difference between the five watermarked apps and the original one for each app shows that AppInk causes very small runtime overhead.

Second, we measure how much time is required to recognize a watermark. This is the time that an arbitrating party needs to verify an app’s originality. For that purpose, we feed these watermarked apps into the extended Android emulator as presented in Section 5.3.5, and measure how much time elapses when AppInk recognizes the watermark values. Table 5.2 shows the average watermark extraction times for the selected watermarked apps. As shown, it takes 6.7 seconds to 28 seconds to verify these apps. Also, a longer watermark value in general requires more time to be recognized, but the difference is small. Compared with the data in Table 5.2, we find that the main time for watermark recognition is spent on the app execution itself. The time differences for the same watermarked apps in these two tables show the execution times dedicated to the watermark extraction, which are from 1.20 seconds to 1.68 seconds. With this scale of time requirement, AppInk’s watermark recognition can be deployed at the current largest app store to handle thousands of app submissions every day.

5.6 Discussion

Our prototype implementation and evaluation have demonstrated the effectiveness of AppInk in preventing the propagation of repackaged Android apps and the deployable capability for general Android app development in practice. In this section, we examine possible limitations of the current prototype and discuss future improvements.

First, AppInk uses a conservative model-based test generation algorithm to generate manifest apps, which maybe not the optimum for watermarking purposes. One possible enhancement is to investigate the latest automatic test case generation methods that have been studied by researchers in the software engineering field. For example, the concolic execution based and GUI ripping based techniques [5, 6, 107] are actively investigated in the software engineering community to enable automatic generation of high-coverage test inputs for Android apps. We plan to study these methods to see if they can be leveraged by AppInk for watermarking purposes.

Second, our AppInk only prototype supports user input events, which are the primary driver for
app functionality, but ignores possible discrete system events, such as short message received, incoming phone call, and various sensor events. We plan to study the working mechanisms of all of these events and explore ways to incorporate them into AppInk.

5.7 Summary

App repackaging is a serious threat to the Android ecosystem, which includes app developers, app store operators, and end users. To prevent the propagation of unauthorized repackaged apps, we propose to adopt a dynamic graph based watermarking mechanism and discuss two usage models where this mechanism is most useful. To make the watermarking mechanism easily integrated into current app development practice and conveniently deployable by relevant parties, we introduce the concept of manifest app, which is a companioning app for an Android app under protection. Based on this we design and implement a tool named AppInk to generate manifest apps, embed watermarks in the apps, and extract the watermarks without any user intervention. Our robustness analysis and practical evaluation against currently available open source tools demonstrate that AppInk is effective in defending against common automatic repackaging threats while introducing a trivial performance overhead.
Chapter 6

Diversifying Intermediate Language for Anti-Repackaging on Android

6.1 Introduction

With the existence of various technical solutions to detect repackaged apps and to protect Android apps from repackaging, app repackaging remains a serious threat to the emerging mobile app ecosystem. Some of the solutions are reactive defenses [31, 60, 123, 154, 155], designed mainly for postmortem detection and jurisdiction and thus cannot prevent apps from being repackaged in the first place. Others do provide proactive defense, but cause serious deployment problems. For example, server-side license verification, introduced by Google and other store operators [71, 122], can be easily bypassed. There even exist automated tools for just that purpose [103–105]. Thus they are usually advised to be deployed with other protective mechanisms, such as obfuscation, to make them difficult to bypass [71]. However, existing Android-based obfuscators [100] are relatively easy to bypass due to the fact that Dalvik bytecode contains rich semantics of an app. Figure 6.1 shows a short snip of code processed by Proguard [100], arguably the most popular obfuscator for Android due to its inclusion in the official Android developer’s guide. Notice that calls to the Android framework APIs are virtually not changed, providing “valuable” information to the repackager of where and how to make modifications. Although it is technically viable to obfuscate these calls with Java reflection and string transformation, it likely will lead to unacceptable overhead because such calls are highly frequent. The watermarking solution, introduced in Chapter 5, although effective and efficient, is mainly a deterrence for app repackaging and needs a trusted authority to arbitrate app’s ownership by extracting the watermark embedded.

In this paper, we propose DIVILAR(DIVersified Intermediate Language for Anti-Repackaging), an Android protection scheme that diversifies the Dalvik bytecode to significantly raise the bar against app repackaging. DIVILAR draws its inspiration from an effective and reliable technology on the traditional desktop systems: virtual machine(VM) based protection, in which the target binary is re-encoded in a
const-string v0, "content://com.example.notepad.provider.NotePad/notes"
invoke-static {v0}, Landroid/net/Uri;->parse(Ljava/lang/String;)Landroid/net/Uri;
mOVE результат object v0
sput object v0, Lcom/example/android/notepad/e;->a:Landroid/net/Uri;

Figure 6.1: Snip of Code Processed by Proguard (only app’s class names are changed).

randomized virtual instruction set and packed together with a specialized interpreter for the instruction set. At run-time, the interpreter translates and executes the protected app. (For brevity, we will call it the guest app.) By encoding the guest app in an unknown instruction set, VM-based protection immediately disables existing tools that expect the original instruction set as input. It thus becomes necessary for the attackers to first reverse-engineer the VM interpreter in order to recover and manipulate the guest app. However, an interpreter could be implemented in a highly convoluted and obscure way. For example, the original QEMU (before TCG) [17] demonstrates how complicated a benign emulator (a type of virtualization) could be. Moreover, virtual instructions can be readily randomized and customized for individual software (or software revisions), further increasing the reverse-engineering efforts. As such, VM-based protection has been successfully applied by both malware authors and developers to protect their “intellectual property” from being analyzed and reverse-engineered [52, 106, 135, 136, 142].

Successful deployments of VM-based protection have been largely limited to native binaries, but not for managed code such as Java Bytecode or Microsoft Intermediate Language (MSIL) [52, 106, 135, 136, 142]: performance overhead caused by the extra layer of instruction interpretation is prohibitive for an interpreter in the managed code, but has little effect over a native interpreter. A native interpreter not only executes faster (because it is executed natively), but also can employ mature technologies such as fast binary translation [13] and just-in-time compilation [98] to improve performance. In contrast, managed code already requires a layer of interpretation/translation. The additional layer of execution can further significantly slow down the system. For example, Proteus [7], a VM-based protection system designed for MSIL, reports from $50 \times$ to $3500 \times$ performance overhead. Such an overhead is even more prohibitive on a mobile platform due to its limited computational power and battery life.

To address this challenge, we observe that Android already has a mature and efficient interpreter for its apps, Dalvik VM. Dalvik VM is loaded as a native library into each app’s address space, which is shared with the app’s own native libraries. The latter thus can freely access and manipulate Dalvik VM at run-time. DIVILAR leverages this accessibility to dynamically hook into Dalvik VM and decodes the virtual instructions right before they are loaded for execution. By doing so, the interpreter for the virtual instructions is merged into the existing execution engine of Dalvik bytecode. This not only significantly reduces the overhead of the extra layer of execution, but also makes DIVILAR much harder to reverse-engineer: compared to interpreters implemented in managed code [7], DIVILAR’s interpreter is implemented as a native binary. Therefore, many coding/obfuscation techniques not available to man-
aged code (e.g., direct memory manipulation) can be employed and a large body of existing obfuscation tools can be directly applied to it.

We have implemented a prototype of DIVILAR. It accepts an Android app as input, transforms its Dalvik bytecode into a randomly generated intermediate language, and wraps the resulting binary together with a lightweight virtual instruction interpreter. DIVILAR is intended to be used by app developers at the last stage of development before releasing the app into public online app stores. DIVILAR protected apps need no special treatment from Android or the app store. They can be released, installed, and executed in the same way as normal Android apps. Our evaluation demonstrates that DIVILAR is robust against common countermeasures, including existing static and dynamic analysis, including those specific to VM-based protection. DIVILAR also has a small and acceptable performance overhead with an average of 15.1% increase to app start time and 9.1% to run time. The extra memory overhead required by DIVILAR is also small, with an average of a 3.1% increase to the private dirty memory space.

In summary, this paper makes the following contributions:

- First, we propose a diversified virtualization-based protection for Android to thwart app repackaging. DIVILAR-protected apps are compatible with the existing Android ecosystem. To the best of our knowledge, it is the first VM-based protection system for Android.

- Second, we design a lightweight in-app hooking mechanism for DIVILAR’s interpreter to composite virtual instruction and Dalvik bytecode execution. Evaluation shows that our prototype interpreter is robust and incurs small performance overhead adequate for everyday usage.

- Third, we have implemented a prototype of DIVILAR. Our evaluation demonstrates that DIVILAR is effective against common countermeasures, including static analysis, dynamic analysis, and particularly those specific to VM-based protection or obfuscation.

6.2 Background

In this section, we introduce key concepts of Dalvik VM, the execution engine for Android apps.

6.2.1 Android App and Dalvik VM

Most Android apps are written in the Java programming language with some components optionally implemented as native libraries. At compiling time, the Java source code is compiled into Dalvik bytecode, which is then assembled into a single Dalvik executable (the .dex file). Each Android app can optionally load native libraries and interact with them through the JNI interface [36]. To release an app, all files of the app are compressed into an apk file that is subsequently signed with the developer’s private key. The developer’s key is usually self-certified without involvement of a central certificate authority.
The signature of an app therefore cannot be used as an indication of trustworthiness of the developer, or prove that the signer is the rightful owner of the app. Instead, Android security model uses the signature to safely allow apps of the same developer to share certain privileges.

Android apps are executed by Dalvik VM, a register-based Java virtual machine designed by Google. Figure 6.2 shows the high-level architecture of Dalvik VM. There are two major components: class loader and execution engine. Class loader is responsible for loading Dalvik classes for execution. Specifically, when a new class needs to be executed, class loader reads its definition from either the app’s \texttt{dex} file which contains all the classes of the app, or from system libraries. The class definition includes information such as fields, methods, and bytecode for each method (except the abstract method). For safety, class loader needs to perform a fairly complicated verification to make sure the class is well-formed and does not violate constrains set by the Java specification [102]. Since DIVILAR uses a completely different instruction encoding than Dalvik bytecode, it is necessary for DIVILAR to bypass the verification of class loader.

Execution engine of Dalvik VM decodes and executes the bytecode of Java methods. Dalvik bytecode has a different encoding scheme than the standard Java bytecode: it is register-based (the standard Java bytecode is stack-based) and has variable lengths (1, 2, 3, or 5 code units, each code unit is 16-bit). For example, the \texttt{add-int} instruction has an opcode of $90_{16}$ and three register operands encoded in 2 code units. By re-encoding an app in a virtual instruction set, DIVILAR will lead existing Android analysis tools (e.g., the \texttt{baksmali} disassembler [3]) to misinterpret the instruction boundaries and thus fail to decode the instruction stream. The original implementation of Dalvik VM executes bytecode through opcode-dispatching, in which it fetches and decodes the bytecode into opcode and operands, and dispatches execution to the corresponding handler for the opcode. Recent versions of Dalvik VM
speed up the execution by supporting just-in-time compilation, where frequently-executed bytecode is compiled into native code and executed natively.

Dalvik VM is shipped as a native library (libdvm.so) and loaded into each app’s address space, which it shares with other app-provided native libraries. The virtual instruction interpreter of DIVILAR is also a native library (libhook.so). At run-time, libhook.so hooks into Dalvik VM and manipulates its internal data structure for its own purposes.

### 6.3 Design

#### 6.3.1 DIVILAR Overview

DIVILAR aims at protecting Android apps from being repackaged by re-encoding them in a diversified virtual instruction set. We have three design goals for DIVILAR:

**Robustness:** although fundamentally it is an arms race, we expect DIVILAR to be robust against existing countermeasures, including commonly available static analyses, dynamic analyses, and those specific to VM-based protection. DIVILAR should also be architecturally flexible to accommodate new countermeasures.

**Compatibility:** DIVILAR-protected apps should be compatible with the current Android ecosystem. Specifically, the developers should be able to release their protected apps in the official Google play store or any third-party app stores, and users can download, install, and execute these apps just like normal apps. Compatibility is the key to the adoption of DIVILAR. Any requirements to change the Android framework will significantly limit the usability of such solutions.

**Performance:** DIVILAR introduces an extra layer of execution to managed code. With a straightforward design (e.g., implementing DIVILAR in the managed code), such a system will lead to prohibitive overhead [7]. Excessive performance overhead will render DIVILAR unsuitable for everyday use. As such, we design a lightweight hooking mechanism to merge virtual instruction execution into Dalvik VM to minimize its performance overhead.

Figure 6.3 shows the overall architecture of DIVILAR. It has four components: virtual instruction selector, bytecode transformer, virtual instruction interpreter, and Apk packager. Virtual instruction
selector decides a virtual instruction set and produces a set of transforming rules (and its inverse) to
describe how to translate Dalvik bytecode into virtual instructions (and vice versa). These two rules are
used by the second and third components to guide conversion between Dalvik and virtual instructions.
Specifically, bytecode transformer applies the transforming rule to an Android app to convert its classes
from Dalvik bytecode to the selected virtual instruction set. The interpreter is an execution engine for
the virtual instruction set. It is customized by the inverse transforming rule so that it can reverse virtual
instructions back to Dalvik bytecode. Moreover, the interpreter is a native library to be bundled together
with the guest app. At run-time, it will be loaded into the app’s address space to execute the guest app.
The last component, apk packager, packs the guest app and its interpreter inside a shell app. The shell
acts as a proxy for the guest app. (It is needed because the guest app is no longer recognizable by An-
droid.) It is responsible for preparing the running environment for the guest app, such as loading the
interpreter. The shell also creates a custom class loader for the guest app, and uses it to load the guest’s
classes for execution. From Android’s point of view, the shell is a well-formed app, while the guest app
and its interpreter are just resources loaded by the shell. As such, a DIVILAR-protected app can pass
the verification and be released, downloaded, and installed in the same way as a regular app.
DIVILAR is architecturally flexible. Various components can have varying levels of design and im-
plementation. For example, the selector defines the conversion between Dalvik bytecode and virtual
instructions. It could use a straightforward one-to-one mapping or implement complicated transforma-
tion by mixing cryptography algorithms as long as the process is reversible. Similarly, the interpreter
can decode virtual instructions in different granularity: one method at a time, one basic block at a time,
or one instruction at a time. Different granularity leads to different trade-off between performance (finer
granularity has higher overhead) and information leakage (finer granularity leaks less information to an
adversary who can monitor the memory of the running app). In the rest of this section we will give more
details about these four components of DIVILAR.

6.3.2 Virtual Instruction Selector

DIVILAR transforms an app encoded in Dalvik bytecode to that in a diversified virtual instruction set.
Consequently, attackers are forced to first reverse-engineer virtual instructions as implemented by the
interpreter, a native binary that can be readily obfuscated. In DIVILAR, each protected app is encoded
in an unique set of virtual instructions decided by virtual instruction selector. The selector produces two
rules: a transforming rule $X$ to convert Dalvik bytecode to virtual instructions, and its inverse $X'$ to
reverse the process. DIVILAR does not pose any restrictions on the selector as long as the conversion
is reversible. For example, it is possible to choose fixed or variable instruction length, apply different
encoding schemes for opcodes and operands, or even use different styles of bytecode (stack-based v.s.
register-based). Different choices of virtual instructions will affect DIVILAR’s capability in resisting
attacks that intend to reverse-engineer the interpreter.
Table 6.1: Example Mapping Rules for Opcode (in hex)

<table>
<thead>
<tr>
<th>Dalvik bytecode</th>
<th>Original opcode</th>
<th>Target opcode</th>
<th>Transforming rule</th>
<th>Inverse rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>add-int</td>
<td>90</td>
<td>7101</td>
<td>90 → 7101</td>
<td>7101 → 90</td>
</tr>
<tr>
<td>invoke-static</td>
<td>71</td>
<td>3202</td>
<td>71 → 3202</td>
<td>3202 → 71</td>
</tr>
<tr>
<td>if-eq</td>
<td>32</td>
<td>9003</td>
<td>32 → 9003</td>
<td>9003 → 32</td>
</tr>
</tbody>
</table>

In our prototype, the selector is designed as a plugin of DIVILAR so that selectors with different levels of sophistication can be readily used and new selector can be created in response to unforeseen attacks. Our prototype implements a linear mapping between Dalvik opcode and virtual opcode. For ease of use by the other modules, the selector generates two mapping rules $X = \{R_1, R_2, ..., R_n\}$ and its inverse $X'$ to formally describe how to translate from and to the Dalvik bytecode. Specifically, each member of $X$ defines how the original Dalvik bytecode is transformed into a randomized virtual instruction. Table 6.1 gives some examples of the mappings for opcode. For example, Dalvik opcode $90h$, an add-int instruction, will be replaced by a two-byte opcode $7101h$. A tool that expects Dalvik bytecode will interpret this two-byte opcode as the invoke-static instruction (opcode $71h$) with $01h$ being the first byte of the operand [127]. Semantically, this “instruction” tries to invoke a static function (specified by 2 following bytes) with zero parameter. This mapping will also shift the instruction boundary around because $90h$ (add-int) has 4 bytes while $71h$ (invoke-static) has 6 bytes instead. Our evaluation shows that even this linear mapping can disable existing dynamic and static analysis tools, including those specific to VM-based protection. The produced set of rules will be used in bytecode transformer and virtual instruction interpreter to guide conversion between Dalvik bytecode and virtual instructions.

6.3.3 Bytecode Transformer

Bytecode transformer converts an Android app (an apk file) encoded in Dalvik bytecode to the instruction set decided by the selector. It first extracts from the app its Dalvik executable file (classes.dex), which contains definitions of all the app’s classes. The transformer then decomposes the dex file into a list of classes and further parses them into fields and methods. A method consists of a method prototype and its associated Dalvik bytecode. At this point, the transformer converts the Dalvik instructions (including its opcode and operands) into virtual instructions guided by the transforming rule $X$. For example, in our linear mapping rule, opcode $90h$ will be converted to $7101h$, and opcode $71h$ will be converted to $3202h$ and so on. As an example of more sophisticated transformation, our prototype also supports an opcode chaining mode in which (the first byte of) the last virtual opcode is xor’ed with the next Dalvik opcode to get the input to the transforming rule $X$. For example, if the last target virtual opcode is $9003h$ (Table 6.1) and the next Dalvik opcode is $E1h$. The input to $X$ is $71h \oplus E1h$,
Figure 6.4: Function Hooking in DIVILAR

and the target opcode will be $3202h$. Such small tweaks are effortless to implement but will make the interpreter harder to reverse-engineer. After transformation, the resulting classes will be re-assembled into a DIVILAR-executable file. This file will be packaged into the final app as a data asset to be loaded and executed by the interpreter, described in the next section.

6.3.4 Virtual Instruction Interpreter

Using virtualization to protect apps is in general an expensive operation. For example, even in the native x86 mode, simple implementations could incur tens of times of overhead [42]. When an extra layer of virtualization is added to managed code, the performance simply becomes unacceptable for any real-world use [7], especially for mobile platforms where computation power and battery life are severely limited. Most Android apps consist of managed code and run on mobile devices such as smartphones and tablets. Therefore, it is critical for DIVILAR to minimize the performance overhead. The overhead comes mainly from the extra layer of bytecode execution. Observing that Android already contains a mature execution engine for Dalvik bytecode (Dalvik VM), DIVILAR tackles this problem by merging its interpreter into Dalvik VM, thus eliminating one layer of execution. More specifically, Android provides a Java virtual machine called Dalvik VM to execute the bytecode of apps. Dalvik VM is a mature intermediate language execution engine with rich features such as just-in-time compilation. Over the years, Dalvik VM has gradually stabilized with few updates. In Android, it is shipped as a native library (libdvm.so) and loaded into the address space of every Android app, where it co-exists with other user-supplied native libraries (Android apps are free to load native binaries.) There exists no security boundary to isolate them (Figure 6.2). Therefore, it is feasible for DIVILAR to load its interpreter as a native library into the protected app, and manipulate Dalvik VM to piggy-back the execution of virtual instructions. The interpreter needs to be written in native code so it can directly operate on Dalvik VM’s memory and issue system calls such as mprotect to change memory attributes.

To facilitate manipulation of Dalvik VM, DIVILAR provides a light-weight hooking mechanism.

---

1This alleviates the concern that DIVILAR is too dependent on the internals of Dalvik VM and thus is less compatible.
(Figure 6.4) similar to Detours [65]. In DIVILAR, hooking points are function calls whose functionality needs to be extended or intercepted (blx origFunc in the figure). The call instruction will be overwritten with another instruction to redirect control flow to a trampoline. The trampoline is a short sequence of ARM instructions. It first saves the current CPU states (caller-saved registers \( r_0 - r_4 \) and the link register \( lr \)) to the stack, and loads the address of the extending function (tgt function in libhook.so) into one of the available registers \( (r_4) \) for execution. After tgt returns, the trampoline calls the original function so the call’s functionality can be maintained and augmented. Finally, the trampoline restores the CPU state and returns back to the call site. Alternatively, we can insert a detour at the beginning of the original function (instead of the call site) to hook the function. We choose the current design because it provides more fine-grained control over when and where to hook a function.

With this light-weight hooking mechanism, DIVILAR can hook into Dalvik VM to intercept and modify its execution in order to composite execution of virtual instructions and Dalvik bytecode. Manipulation of VM states at run-time requires clear understanding of its internals and careful planning. Figure 6.5 shows the major components of Dalvik VM: a class loader and an execution engine. Both components can operate on the shared run-time data such as method definitions and the heap. The execution engine itself has a number of components, such as the bytecode interpreter (to execute bytecode), JIT compiler (to compile hot spots into native instructions), and class linker (to link classes from different Java libraries). Architecturally, DIVILAR can accommodate different designs of virtual instruction execution. For example, an implementation can provide a complete execution engine (possibly based on the Dalvik VM source code) and totally bypass Dalvik VM. Our prototype implements a pre-execution
decoding scheme. More specifically, it hooks into functions that execute a method and decodes the method by reversing the opcode and operand mapping before its execution. In addition to these functions, we also need to modify class loader (so it knows how to load the guest app) as well as the meta-data in the method area. Section 5.4 contains details about our prototype. Figure 6.6 shows the logical architecture of Dalvik VM after the interpreter (\texttt{libhook.so}) is loaded. In the figure, \texttt{libhook.so} is responsible for the loading of the guest app’s classes because these classes are encoded in virtual instructions and cannot pass the sanity check of Dalvik’s class loader. DIVILAR’s execution engine hooks into and leverages Dalvik’s execute engine to run the app. Technically, the interpreter contains the complete information required to recover the original app. It is thus important to prevent it from being reverse-engineered. To this end, various obfuscation technologies for native binaries can be readily applied to \texttt{libhook.so}, such as packer/unpacker [58, 108].

### 6.3.5 APK Packager

One of our design goal for DIVILAR is compatibility so that DIVILAR-protected apps do not require special install-time processing or modifications to the Android framework. As such, APK packager, the last component of DIVILAR, wraps all components necessary to execute the guest app into a single shell app. The interpreter (\texttt{libhook.so}) and the guest app are both contained in the shell as data assets. The shell app is a simple wrapper of the guest app, synthesized by APK packager. From Android’s point of view, the shell app is the main body of the final app. Therefore, it can pass the install-time and run-time verification by the Android system. The shell app inherits the permissions required by the guest app so it can execute the guest app with enough permissions. It also needs to provide a wrapper for every entry point in the guest app, such as activities, content providers, and services defined in the guest app’s manifest file [79]. These entry points are automatically loaded by Dalvik VM when starting an app. At run-time, when the shell’s entry points are called, they simply load the corresponding guest app classes.
Initialize class loader for guest app \rightarrow \text{Setup in-app hooks to Dalvik VM} \rightarrow \text{Pre-load some critical classes} \rightarrow \text{Init main activity & execute from its onCreate}

Shell App and DVM Initialization \hspace{1.5cm} \text{App Execution}

Figure 6.7: Execution Flow of the Protected App

into memory and dispatch them for execution. Note that this loading process does not decode virtual instructions. Instead, they are decoded only when necessary by libhook.so. After collecting all the required components, APK packager reassembles them into a single apk file which can be released, downloaded, and installed as normal apps.

The final app produced by DIVILAR consists of the shell, the guest app, and the interpreter. Figure 6.7 shows the run-time execution flow of the app. The shell first initializes the execution environments for the guest app. Specifically, it creates a custom class loader for the guest app’s classes, loads the interpreter in the memory and hooks it to Dalvik VM, and finally pre-loads some critical classes of the guest app because Android expects these classes to be loaded upon returning from initialization. These steps need to be performed before executing the first instruction of the guest app. For this purpose, the shell app extends a special Android class called \text{android.os.Application}. This class (or its subclass) is guaranteed to be executed just after the (shell) app’s classes are loaded and before the first bytecode of the main activity is executed.

### 6.4 Implementation

We have implemented a prototype of DIVILAR based on Android 4.0.4. Our prototype is a standalone Java application. It accepts an existing app as the input and generates a functionally-equivalent protected app. DIVILAR is intended to be used at the last stage of app development. The resulting app can be released in the online app stores for users to download and install. As mentioned earlier, the DIVILAR architecture can accommodate different levels of sophistication in the implementation. In this section, we describe details of our prototype in the order of DIVILAR’s four components: instruction selector, bytecode transformer, the interpreter, and apk packager.

Our prototype adopts a linear mapping for Dalvik bytecode’s opcode and operands (Section 6.3.2). To guarantee that each protected app is encoded differently, the instruction selector uses a random number to permute the opcode/operand mapping. This generates two rules: the transforming rule $X$ and its inverse. For example, $X = \langle 7503, 83, \ldots \rangle$ specifies that opcode $01h$ will be converted to $7503h$, opcode $02h$ will be converted to $83h$, and so on. These rules are subsequently used to guide conversion
between Dalvik and virtual instructions for the second and third components, bytecode transformer and the interpreter.

Bytecode transformer re-encodes the guest app in the virtual instruction set as specified by the transforming rule $X$. Our implementation utilizes the library of baksmali/smali [3], an open-source Dalvik disassembler/assembler. Specifically, the guest app is first parsed into a list of classes with their methods disassembled to the string format. Bytecode transformer then reassembles these classes into a dex file except that the opcode/operands have been adjusted according to rule $X$ (file org/jf/dexlib/Code/Opcode.java of the baksmali library is changed to write out the custom opcode and operands). For example, the bytecode 01h is first parsed into its readable format (move in this case). DIVILAR then converts move to the new opcode of 7503h when assembling the file. All classes of the guest app will be assembled together into a single dex file, and further packed into the final app as a data asset for the interpreter to execute.

The next step of the prototype generates a virtual instruction interpreter customized by rule $X'$. At run-time, the interpreter hooks into Dalvik VM in order to composite the virtual and Dalvik instruction execution. The interpreter is a native binary written in the C++ programming language. When loaded for execution, it first uses system call mprotect to make Dalvik VM writable so that it can be hooked. It then searches the code section of Dalvik VM for call sites of some specific functions, as determined by the hooking strategy. Each of these call sites will then be replaced by a call to the corresponding trampoline, as shown in Figure 6.4. A wide range of hooking schemes can be employed. In our prototype, we choose to extend the class initialization module, which is responsible for bytecode loading and method initialization (in file vm/oo/Class.cpp and vm/oo/Object.cpp). We add to it the pre-execution conversion of virtual instructions to Dalvik bytecode. To reduce the risk of the decoded methods being extracted from memory, our prototype flushes the Dalvik code cache randomly from time to time.

At the last step, APK packager assembles the guest app and the interpreter into a final app that can be publicly released later. It first parses the manifest file (AndroidManifest.xml) of the original app to get the list of requested permissions, entry points, and other resource files. It then synthesizes a wrapper (the shell) for the guest app. The shell inherits the requested permissions from the original app, and has a wrapper class for each entry point. It is also responsible for loading and executing the guest app, a task performed in its subclass of android.os.Application because it is guaranteed to be executed before any other classes.

### 6.5 Evaluation

DIVILAR aims to protect Android apps from being repackaged by applying VM-based protection. In this section, we first analyze DIVILAR’s robustness against existing static and dynamic analyses, and countermeasures specific to VM-based protection in particular [30, 51, 130, 134]. We also evaluate the performance and memory overhead introduced by DIVILAR.
6.5.1 General Analysis

DIVILAR re-encodes Android apps in a secret and diversified virtual instruction set. The guest app is wrapped as a data asset in the resulting app together with the interpreter for these virtual instructions. The interpreter hooks into the complicated Dalvik VM to composite execution of virtual and Dalvik instructions. By introducing this additional layer of indirection, existing tools that expect Dalvik bytecode will immediately be disabled. Attackers have to shift their focus from understanding Dalvik bytecode into reverse-engineering the interpreter first (presumably, the interpreter contains adequate information about the virtual instructions). In this sub-section, we will analyze DIVILAR’s protection strength against app repackaging.

**Rooted in native code:** managed code is in general harder to obfuscate than native code because it contains rich semantics of the app. For example, intermediate languages like Java bytecode often use late binding, in which the target of a call is resolved at run-time by name. As such, call instructions in an intermediate language normally contain a reference to the method name as shown in Figure 6.1. Moreover, managed code has numerous run-time and compile-time checks to ensure program safety. This significantly limits the operations that can be performed by the program. Operations such as complex pointer arithmetic or direct memory access are almost always disallowed. Native code instead runs directly on the physical machine and has few restrictions in operations and program structures. Native code thus has more freedom in the support of obfuscation. Particularly, in native code, code and data can be mixed together; instructions can directly access any memory mapped in the program; a program can also mutate itself with the self-modifying code. Obfuscation in native code can leverage these features to make it more reliable. For example, Giffin et al. [53] propose to strengthen software self-checksumming using self-modifying code; Jacob et al. [91] leverage overlapped instructions in the x86 architecture to implement a stronger tamper-proof solution. These technologies make obfuscated native binaries more difficult to reverse-engineer, usually requiring laborious human effort.

Compared with earlier virtualization based protection for bytecode (e.g., Proteus [7]), DIVILAR has its robustness rooted in the native code. Instead of implementing the interpreter in Dalvik bytecode, the interpreter is a native executable. This not only reduces the overhead, but also allows advanced obfuscation technologies [53, 91] to be applied, making the interpreter harder to reverse-engineer and thus significantly raising the bar against app repackaging.

**Varied virtual instruction selections:** architecturally, DIVILAR can accommodate a wide range of virtual instruction selections as long as it can be reversed. The instruction selector works as a plugin of DIVILAR to support different encoding schemes, such as fixed-length vs. variable-length instructions, stack-based vs. register-based instructions, or even overlapped instructions [91]. Moreover, DIVILAR randomizes the virtual instruction set for each individual app. Breaking one protected app thus does not necessarily lead to compromise of other protected apps. In our prototype, we choose a linear mapping.
unknown opcode encountered - 73. Treating as nop.

UNEXPECTED TOP-LEVEL EXCEPTION:
org.jf.dexlib.Util.ExceptionWithContext: 72
   at org.jf.dexlib.Util.ExceptionWithContext(ExceptionWithContext.java:54)
      ....
   at org.jf.dexlib.Section.readFrom(Section.java:143)
   at org.jf.dexlib.DexFile.<init>(DexFile.java:431)
   at org.jf.baksmali.main.main(main.java:280)
Caused by: java.lang.ArrayIndexOutOfBoundsException:72
   ....
   ... 6 more
Error occurred at code address 140
code_item @0x784

Figure 6.8: Exception When Applying baksmali on a Transformed App

for virtual instructions. The combination of such a mapping \(^2\) is sufficiently large so that the brute-force attack (to try every possible combinations) is basically infeasible. Meanwhile, our prototype can confuse a parser of Dalvik bytecode with shifted instruction boundaries. This makes frequency analysis ineffective because it relies on a parser to decode the app first (Section 6.5.4). Therefore, even with this simplified virtual instruction selection, our prototype provides a strong protection against existing countermeasures.

**Diverse interpreter design:** to reduce overhead, DIVILAR hooks into Dalvik VM in order to composite virtual and Dalvik instruction execution. A wide range of interpreter design (hence hooking schemes) can be supported. For example, it is possible to even write a standalone interpreter and completely bypass Dalvik VM. Different interpreter design needs to modify different components of Dalvik VM (Figure 6.5). Our prototype implements a pre-execute decode scheme in which virtual instructions are reverted back to Dalvik bytecode before execution. An adversary monitoring the memory of the app might be able to extract the recovered bytecode. To reduce the time window of possible exposure, our prototype will randomly flush the code cache. To further reduce the window, we may hook into code that executes instructions in a finer granularity, such as basic block level or instruction level. To protect the interpreter from reverse-engineering, the interpreter should be obfuscated, as mentioned before. A large number of existing obfuscation schemes for native binaries can be applied.

In addition, with the interpreter implemented in native code, performance overhead will not be a concern for DIVILAR since a variety of optimization techniques can be applied \([4, 12, 20, 145]\), such as ahead-of-time compilation or adaptive optimization.

**Compatible with other schemes:** DIVILAR encodes the guest app in a diversified virtual instruction set. This kind of transformation is compatible with a wide variety of other protection mechanisms.

\(^2\text{There are } 256! > 8.578 \times 10^{506} \text{ combinations to map an one-byte-long opcode set to another one-byte-long opcode set.}
\text{The combination to map an one-byte-long opcode set to a two-byte-long opcode set is } 1024!/256! > 10^{1024}\)
For example, AppInk [153] embeds watermarks into an app to identify the source of an app. DIVILAR can be applied to the app after watermarks are embedded. Obfuscation techniques can also be applied before or after DIVILAR. For example, Java based obfuscation can be attempted first, and then DIVILAR is applied, or DIVILAR is used first and native code obfuscation can be used to obfuscate the interpreter in the resulting app. Moreover, DIVILAR aims to provide an strong preventive mechanism for apps. It does not provide services such as integrity check in itself. These services can also be combined with DIVILAR. For example, self-checksumming is often used to ensure program integrity. Due to various constrains of managed code, self-checksumming is challenging to implement in managed code. With DIVILAR, it can be implemented in the native code and be used at various stages of Dalvik VM execution. This would make self-checksumming for Android not only feasible but also more reliable against adversaries.

In the following part of this section, we analyze DIVILAR’s robustness against various specific static and dynamic analyses, including experiments with popular tools frequently used in Android app repackaging.

6.5.2 Static Analysis

To repackage an app, attackers often apply some form of static analysis to understand how the app is organized and where to make necessary modifications. All these tools include some form of disassembler to parse the app into machine- or human-readable format, such as the mnemonic representation of Dalvik bytecode. Baksmali [3] probably is the most popular disassembler for Android apps. Its source code has been embedded in many other Android analysis/reverse-engineering tools [54, 126]. Meanwhile, many static analysis tools for Android, such as Dare [116] and smali2java [54], re-target Android apps to Java bytecode (Java bytecode is stack-based while Dalvik bytecode is register-based) in order to leverage a rich set of existing Java analysis tools [144]. In this subsection, we will evaluate DIVILAR’s resilience against two representative static analysis tools: baksmali [3] and Dare [116]. Specifically, we encode seven Android apps into virtual instructions, and apply these two tools on the resulting apps to observe their behaviors.

Baksmali is an open-source Android disassembler. Its companion project, Smali, is a corresponding assembler. Baksmali and Smali are frequently used to manually repackage apps [96]. Baksmali provides a library that can reliably parse Dalvik executable files [127] and further disassemble them into the mnemonic format. The source code of Baksmali has since been embedded in many other tools (including DIVILAR). For all seven of these apps, Baksmali reports an error message of “unknown opcode encountered - nn. Treating as nop” and throws an unexpected top-level exception. Figure 6.8 shows an encountered exception when we apply Baksmali to one of the guest app. Baksmali and other Android analysis tools expect Dalvik bytecode as inputs. By re-encoding Android apps in virtual instructions, DIVILAR has changed the semantic of opcode seen by these tools and shifted instruction boundaries.
around. However, these tools still interpreter the guest app according to the Dalvik bytecode format. This usually leads to unknown opcodes, as shown in Figure 6.8.

To leverage a wide variety of existing static analysis tools for Java bytecode, some analysis tools opt to re-target Android apps into Java bytecode. Dare [116] is one of the most effective tools for this purpose, which is reported to work correctly on more than 99.99% of 262,110 Dalvik classes. Since the very first step of Dare is also to parse the dex file, it fails with a similar symptom as Baksmali. Figure 6.9 shows the error message reported by dare on the same guest app.

The above experiments show that DIVILAR can immediately render these existing Android disassemblers and analysis tools ineffective because they all depend on parsing the well-formed Dalvik bytecode. These tools include decompiler, slicer, control flow analyzer, data flow and data dependency analyzer [3,116,126,144]. For these tools to function correctly, it is necessary to first recover the original Dalvik bytecode by reverse-engineering the interpreter.

### 6.5.3 Dynamic Analysis

Dynamic analysis executes the app under investigation in a monitored environment, such as a virtual machine or an emulator, to observe its behaviors. Dynamic analysis has been widely used in malware analysis and program development. In dynamic analysis, the monitor can be placed either inside or outside of the run-time environment of the target app. The former has the advantage of direct access to system/app states, but its result may be disturbed or even tampered with by the target app. While the latter gains the tamper-resistance, but faces the “semantic gap” challenge [23, 94], in which the monitor has to deduce app states given only externally observable data such as a memory dump [94, 149]. It is challenging to apply this out-of-box approach to analyze Android apps because we need to reconstruct both operating system states and Dalvik VM states due to the two layers of virtualization [149]. DIVILAR makes the problem even more challenging by adding another layer of indirection: the monitor needs to infer states of the operating system, the virtual instruction interpreter, and Dalvik VM.

Our current prototype adopts a pre-execution decoding scheme. An adversary who continuously monitors the app’s memory might be able to extract the bytecode of the original app. Such powerful attacks can be (partially) addressed by either limiting the amount of restored bytecode (e.g., from method to basic block) and shortening the time window of exposure, or by detecting whether the app is run in a
virtual environment (the Android emulator) and refusing to run if so.

### 6.5.4 Virtualization Specific Analysis

Virtualization-based protection/obfuscation has long been employed by malware to hide its malicious behavior. A number of counter-measures have been proposed to detect or prevent such malware. For example, some systems aim to reverse-engineer the VM and use this information to infer semantics of individual virtual instructions [130, 134]. Coogan et al. propose an inside-out approach [30], which intends to identify instructions that may affect the observable behavior of the wrapped code (i.e., system calls). Ghosh et al. propose to replace the protecting VM with an attacking VM in order to render the application amicable to analysis and modification [51]. Frequency analysis [146] is often used to break classical ciphers by studying the frequency of letters or groups of letters to disclose the mapping between plaintext and ciphertext. It may potentially be used to infer the mapping between Dalvik bytecode and virtual instructions. In the rest of this sub-section, we discuss DIVILAR’s robustness against these virtualization-specific attacks.

**Reverse-engineering VM** Static analysis in general will not work effectively against virtualization based protection. However, determined attackers can use dynamic analysis or hybrid analysis to recover considerable information about the virtual machine or the virtual instructions. Recently, a few approaches are proposed to automatically reverse-engineer the virtual machine, and then use the recovered semantics of virtual instruction to understand the original app [130, 134]. However, to precisely and completely recognize and identify a virtual machine’s virtual instructions is hard in both theory and practice. As such, existing approaches limit the scope by, for example, assuming that the VM works in the loop of fetch-decode-dispatch. In the case of DIVILAR, these assumptions do not hold, thus making these approaches ineffective: Dalvik VM has a complex structure with about 104,000 lines of C++ code and 99,000 lines of assembly code. Moreover, Dalvik VM supports many different execution modes, including portable, fast, and JIT modes. Through taking more diversity into the virtual instruction set selection and virtual interpreter design, these execution modes can be extended by DIVILAR in many possible ways, and thus further raise the bar against reverse engineering analysis.

**Inside-out approach** A recent work proposes an inside-out approach [30] that complements the VM reverse-engineering approach. Instead of recovering all instructions of the wrapped app, this approach aims at identifying instructions that interact with the operating system and instructions that may affect these instructions. It is assumed that these instructions together approximate the behavior of the original code, while other instructions are considered to be uninteresting. This approach can help malware analysts to gain a better understanding of the malware under examination without being overwhelmed by details of all the instructions. However, a partial understanding of the virtual instructions is not enough to make meaningful modifications to the guest app and repackage it. In particular, any executable additions to the guest app need to be encoded in the same virtual instructions as the app. This is not possible
without a complete understanding of the virtual instruction set first.

**VM replacement attack** VM replacement attack is proposed to subvert virtualization-protected apps by replacing the protecting VM with an attacking VM, within which the app’s execution can be monitored and analyzed. However, this attack is not effective against systems where virtual machine is sufficiently anchored to the execution environment [51]. DIVILAR’s in-app hooking mechanism allows it to deeply hook into Dalvik VM and merge the execution of virtual and Dalvik instructions. This tight coupling with the underlying execution environment makes VM replacement attacks ineffective against DIVILAR. In addition, DIVILAR can identify whether the underlying VM has been modified or whether an attacking VM (specifically, its code introspection framework) has been loaded into the app’s address space. Essentially, the VM replacement attack in this case becomes an in-VM monitor and thus is subject to tampering [94].

**Frequency analysis** Frequency analysis is often used to break classic cipher where a plaintext letter is mapped to one ciphertext letter. By studying the frequency of letters or groups of letters, the analyst can deduce the mappings between the plaintext letters and ciphertext letters [146]. Our prototype of DIVILAR also uses a mapping table to convert Dalvik bytecode into virtual instructions. However, our system is not susceptible to frequency analysis. Specifically, by simply replacing one letter with another, a counting tool can easily count the frequency of plain-text and cipher-text in a classic cipher. However, it does not know how to count the instruction frequency for virtual instructions without the knowledge of the virtual instruction set first. In fact, such tools would require a parser to decode instructions, which has been shown to be ineffective against DIVILAR (Section 6.5.2).

### 6.5.5 Performance and Memory Overhead Evaluation

In this section we present the experimental results to evaluate the performance and memory overhead introduced by DIVILAR. All the experiments are conducted on an emulator running Android v4.0. Each experiment is performed 40 times, and average results with 95% confidence intervals are reported in the tables below.

In the performance overhead evaluation, we measure both start-time and run-time overheads, two overheads that matter most to end users. We choose five sample apps from the Android SDK as our target apps: jet boy, notepad, contact manager, multi-resolution, and honeycomb gallery. We select these apps because their source code are freely available and we can insert probing points into the apps to measure the time required to execute certain operations.

We first measure how DIVILAR affects the app start up, from launching the app to the appearance of the main activity on-screen. To prepare an app for execution, the Android framework performs a series of steps such as loading Dalvik VM. DIVILAR adds to the start-time in order to prepare execution of virtual instructions. Specifically, the guest app is wrapped in a shell app. Android launches the shell first which in turn loads the guest app. To measure the start time, we use the `logcat` command [76] of
Table 6.2: App Start Times and Extra Delays

<table>
<thead>
<tr>
<th>App Name</th>
<th>Original App (milliseconds)</th>
<th>DIVILAR App (milliseconds)</th>
<th>Extra Delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contact Manager</td>
<td>1395 ± 128</td>
<td>1533 ± 166</td>
<td>9.9%</td>
</tr>
<tr>
<td>HoneyComb Gallery</td>
<td>1539 ± 98</td>
<td>1718 ± 155</td>
<td>11.6%</td>
</tr>
<tr>
<td>Jet Boy</td>
<td>3471 ± 225</td>
<td>4206 ± 378</td>
<td>21.1%</td>
</tr>
<tr>
<td>Multi Resolution</td>
<td>1654 ± 143</td>
<td>2030 ± 212</td>
<td>22.7%</td>
</tr>
<tr>
<td>Note Pad</td>
<td>1592 ± 187</td>
<td>1753 ± 133</td>
<td>10.1%</td>
</tr>
</tbody>
</table>

Table 6.3: Runtime Action Times and Extra Delays

<table>
<thead>
<tr>
<th>App Name</th>
<th>Original App (milliseconds)</th>
<th>DIVILAR App (milliseconds)</th>
<th>Extra Delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contact Manager</td>
<td>51.3 ± 4.7</td>
<td>54.2 ± 6.1</td>
<td>5.7%</td>
</tr>
<tr>
<td>HoneyComb Gallery</td>
<td>247 ± 21</td>
<td>293 ± 27</td>
<td>18.6%</td>
</tr>
<tr>
<td>Jet Boy</td>
<td>1990 ± 127</td>
<td>2019 ± 178</td>
<td>1.5%</td>
</tr>
<tr>
<td>Multi Resolution</td>
<td>483 ± 51</td>
<td>557 ± 47</td>
<td>15.3%</td>
</tr>
<tr>
<td>Note Pad</td>
<td>15.9 ± 1.1</td>
<td>16.6 ± 0.9</td>
<td>4.4%</td>
</tr>
</tbody>
</table>

Android Debug Bridge [73]. The verbose logging level of logcat is used so that it shows the elapsed time from app launching to the appearance of the first app view. Table 6.2 shows the average overhead for both the original apps and DIVILAR processed apps. The extra overheads caused by DIVILAR are shown in the last column. They range from 9.9% to 22.7% with an average of 15.1%.

Next, we measure the overhead of DIVILAR to a running app by adding probe points to log its execution time. Specifically, we first examine their source code to select code ranges where no user interaction is required (so as to exclude the impact of uncontrollable user interaction), and manually insert a call to android.util.Log before and after each of these selected code ranges. We then rebuild these apps and use DIVILAR to generate their corresponding protected apps. Table 6.3 reports the average overhead for both the original apps and the DIVILAR processed apps. The extra overheads caused by DIVILAR are shown in the last column. They range from 1.5% to 18.6% with an average of 9.1%. DIVILAR has a fixed overhead for each method, i.e., the time to translate the method from virtual instructions to Dalvik bytecode. The run-time of a method, on the other hand, is determined by dynamic program states such as branch conditions and loops. For example, a method could execute only a failed branch in one run and execute hundreds of times in a loop in another run. Overall, our interpreter design is efficient and end users will not perceive much delays or stuttering. Compared with the high performance overhead brought by Proteus [7] (30 times to 5,000 times), DIVILAR significantly reduces the extra overhead.
Table 6.4: Private Dirty Memory Sizes for Each App

<table>
<thead>
<tr>
<th>App Name</th>
<th>Original App (kilobytes)</th>
<th>DIVILAR App (kilobytes)</th>
<th>Extra Memory Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contact Manager</td>
<td>3688 ± 103</td>
<td>3823 ± 140</td>
<td>3.7%</td>
</tr>
<tr>
<td>HoneyComb Gallery</td>
<td>6486 ± 478</td>
<td>6648 ± 472</td>
<td>2.5%</td>
</tr>
<tr>
<td>Jet Boy</td>
<td>8525 ± 613</td>
<td>8542 ± 617</td>
<td>0.2%</td>
</tr>
<tr>
<td>Multi Resolution</td>
<td>4903 ± 288</td>
<td>5031 ± 315</td>
<td>2.6%</td>
</tr>
<tr>
<td>Note Pad</td>
<td>3,691 ± 164</td>
<td>3,926 ± 158</td>
<td>6.4%</td>
</tr>
</tbody>
</table>

Android platform is memory efficient in that it saves a lot of memory by forking a main zygote process at the system start time and sharing memory pages with later forked app process. The memory space of Dalvik native library (libdvm.so) is also shared due to this mechanism. Copy-on-write will be used to ensure that a process cannot modify the memory of other processes. Now that DIVILAR will modify libdvm.so, the modified part has to be copied to the app’s own process space. This brings some extra memory overhead. What is more, the newly added native library (libhook.so) needs extra memory space as well. In the memory overhead evaluation, we measure the memory size as required by the above samples apps, particularly the size of private dirty memory, which is exclusively required by the tested apps. To calculate the extra overhead as required by DIVILAR, we report the size of private dirty memory for each of these sample apps, both before and after being processed by DIVILAR. Table 6.4 presents these result. The extra memory overheads are shown in the last column. They range from 0.2% to 6.4% with an average of 3.1%, which is reasonable. The reason is that DIVILAR prototype only touches a very small part of Dalvik VM, and only the individual pages inside libdvm.so that are modified by DIVILAR will be copied into app’s private space. The newly added native library (libhook.so) is also compact. Its size is close to 24k bytes in our prototype. The extra memory requirement is thus small.

6.6 Discussion

In this section, we discuss implications and possible improvements of DIVILAR and, in particular, our prototype.

In essence, DIVILAR is an obfuscation technology used to protect the Android app from repackaging. Although the technology itself is neutral, obfuscation has long been used by malware authors to evade detection by anti-malware software. Many security researchers and practitioners share the opinion that benign software (Android apps in this case) should not use obfuscation so that its behaviors can be easily analyzed or even formally verified. Unfortunately, this is not an option for anti-repackaging. Most apps are repackaged and distributed in third-party online app stores [155]. These stores often lack regulations against app piracy (some even benefit from it), and many are located in other countries. It is thus difficult for developers to resolve the issue through jurisdiction. Instead, the developers have
to enforce their apps with self-defense. App repackagers would try to defeat or bypass these defense mechanisms. It becomes an arms race between developers and app repackagers. For example, Google suggests to use string and other obfuscation techniques to prevent its server-side license verification from being disabled or bypassed [71]. In addition, VM-based protection has been previously proposed to protect applications [52, 106, 135, 136, 142].

Although not an architectural limitation, our prototype has a relatively simple design: a linear mapping of opcode/operand is used to select virtual instructions, and the interpreter implements a pre-execution decoding mode. Our initial experiments and analysis show that this design works reliably against existing static and dynamic countermeasures, including those specific to virtualization-base protection. Nevertheless, the design can be enhanced by additional techniques to thwart analysis. In the future, we will study ways to fortify our current design, particularly, how to formally measure the strength of obfuscation technologies and how to maximize their strength given a certain performance overhead.

In its current operation model, developers use DIVILAR to process their apps before releasing them to the online app store. Thus, all installations of the app are the same across all its users (although different apps will have a different virtual instruction set). In the future, we will study methods to securely perform per-installation obfuscation, for example, to re-encode the app at its first run. In this way, each installation will have a different encoding, possibly tied to the user’s device. This would provide a stronger protection because every installation of the app is different.

DIVILAR hooks into Dalvik VM to composite execution of virtual and Dalvik instructions. Like other such software [63], compatibility with different revisions of the host software is a concern. Dalvik VM, the host of DIVILAR, has become relatively stable since Honeycomb (Android 3.0). Our prototype further alleviates the problem by avoiding hard-coding any specific hooking points and locating them through pattern matching instead. Under the unlikely circumstances that Dalvik VM is changed significantly and frequently, we could implement a self-contained execution engine (possibly based on the source code of Dalvik VM) and completely bypass Dalvik VM. This will improve the compatibility but may lose the benefit of updates made by Google.

Lastly, DIVILAR implements the VM-based protection for Android apps. From another point of view, VM-base protection can be considered as a generation of instruction set randomization [14]. For example, in addition to randomizing the instruction set, VM-base protection can provide a randomized view of memory. Our current prototype focuses on the randomization of Dalvik bytecode. In the future, we will experiment with randomization of other components in VM-based protection as well.

6.7 Summary

App repackaging remains a serious threat to the Android ecosystem and the emerging mobile economy model. To thwart repackaging, we adopt virtualization-based protection to enable the app’s self-defense. To this end, we design an effective solution called DIVILAR which hooks into Dalvik VM
to efficiently composite the virtual instruction and Dalvik bytecode execution. Our evaluation demonstrates that DIVILAR is robust against existing static and dynamic analyses including those specific to virtualization-based protection or obfuscation. Our prototype incurs a small performance and memory overhead that likely is not perceivable to end users. In conclusion, DIVILAR is an effective and promising technology to enable app self-defense and prevent app repackaging in particular.
Chapter 7

Conclusion and Future Work

In this dissertation, we have presented and described the emerging app repackaging threats and potential defenses. Particularly, DroidMOSS presents the first systematic study of app repackaging threat for people to understand its extent. PiggyApp describes a fast and scalable solution to detect piggybacked apps (the most serious category of repackaged app). AppInk presents a watermarking-based mechanism to prove the app ownership, and thus provides an effective solution to deter app repackaging. DIVILAR describes a diversified virtualization-based protection for Android to thwart app repackaging.

In the aspect of app repackaging threat, DroidMOSS and PiggyApp show that app repackaging is a serious and real threat to the prosperous app based economy. Evaluations also show that they are effective and efficient solution to help analyst detect repackaged apps among large data set. In the defensing side, through modifying the app source code, AppInk embeds a developer selected watermark into the running state of the app under protection. When an app is suspected to be repackaged app, the original author can use an automatically generated manifest app to drive the suspect app and extract the watermark from its running state. In this way, attackers can’t deny the app ownership and thus are deterred to repackage these AppInk-protected apps. DIVILAR, instead, directly works on the bytecode of the original app and transforms it into a new intermediate language. Since the new intermediate language can’t be understood by current attacking tools that are based on the Dalvik bytecode semantic, it easily defeats these attacking tools in the first place. Through a carefully designed in-app hooking mechanism, DIVILAR incurs small overhead to the app under protection, reducing the usually prohibitively high extra overhead to a much lower level. To evaluate these two defensing mechanisms, we conduct a series of experiments and analyses, demonstrating that AppInk is effective in proving app’s ownership and DIVILAR is effective in thwarting attacker’s app repackaging attempt. Performance evaluations show that both mechanisms incur small overhead to the apps under protection, and thus are deployable for real world usage.

Based on the insights we gained in working on these four pieces of work, we propose three research directions as future work.
• **Automatic app repackaging detection** Both DroidMOSS and PiggyApp rely on the existence of original apps, and thus require a large app set that include as many original app as possible. Also both solutions generate a similarity score to indicate how similar two apps are, and sometimes need analyst’s further analysis to finally determine the repackaging relationship. We propose future work of continuously collecting as many apps as possible, and exploring the combination of different detection mechanisms to increase the accuracy of app repackaging detection, and also to reduce the reliance on human analyst work.

• **Anti-evasion app repackaging detection** As attackers notice the existence of app repackaging detection solution, it is for sure that they will develop or leverage some evasion techniques to avoid detection. For example, different kind of obfuscation techniques might be used to reduce the detection rates of our systems. We propose future work to investigate into currently available obfuscation techniques to see how they will impact our current app repackaging detection solutions, and how we should improve our detection mechanisms with the existence of these evasion techniques. One potential improvement is to use hybrid detection techniques and assign different weights to various techniques to comprehensively assess the repackaging relationship of the apps under study.

• **Stronger protection mechanism for Android app** Working on different levels and protecting app from repackaging at different stages, both AppInk and DIVILAR are easily deployable and incur small overhead for the apps under protection. However, they also have some limitations. For example, there is no quantitative analysis to show how strong they are, and the implementation prototypes can be further fortified as well. We propose future work to fortify the current design solutions, and to explore formal or quantitative mechanisms to measure how reliable and strong these protection mechanisms are at defending against currently existent repackaging threats. One potential solution is to explore the integration of different techniques at different levels to protect smartphone apps from being repackaged at different stages.
REFERENCES


