



THE HONG KONG
POLYTECHNIC UNIVERSITY

香港理工大學

Pao Yue-kong Library

包玉剛圖書館

Copyright Undertaking

This thesis is protected by copyright, with all rights reserved.

By reading and using the thesis, the reader understands and agrees to the following terms:

1. The reader will abide by the rules and legal ordinances governing copyright regarding the use of the thesis.
2. The reader will use the thesis for the purpose of research or private study only and not for distribution or further reproduction or any other purpose.
3. The reader agrees to indemnify and hold the University harmless from and against any loss, damage, cost, liability or expenses arising from copyright infringement or unauthorized usage.

IMPORTANT

If you have reasons to believe that any materials in this thesis are deemed not suitable to be distributed in this form, or a copyright owner having difficulty with the material being included in our database, please contact lbsys@polyu.edu.hk providing details. The Library will look into your claim and consider taking remedial action upon receipt of the written requests.

CROSS-LAYER DYNAMIC ANALYSIS
OF ANDROID APPLICATIONS

LEI XUE

Ph.D

The Hong Kong Polytechnic University

2018

The Hong Kong Polytechnic University
Department of Computing

Cross-Layer Dynamic Analysis of Android Applications

Lei Xue

A thesis submitted in partial fulfilment of the requirements
for the degree of Doctor of Philosophy

July 2017

CERTIFICATE OF ORIGINALITY

I hereby declare that this thesis is my own work and that, to the best of my knowledge and belief, it reproduces no material previously published or written, nor material that has been accepted for the award of any other degree or diploma, except where due acknowledgement has been made in the text.

_____ (Signed)

Lei Xue _____ (Name of student)

Abstract

Android has become the most popular mobile operating system, and billions of Android applications (apps) have been downloaded from the both official and various third-party markets. Unfortunately, not all apps are benign or well designed, and understanding the behaviours of Android apps is essential for analyzing and detecting malicious apps. To achieve this goal, various static bytecode analysis and dynamic behavior analysis approaches have been proposed. This thesis focuses on dynamic analysis because static analysis could be impeded by the dynamic features of programming languages or the protection mechanisms adopted by apps.

Although a number of dynamic analysis approaches have been proposed to monitor the behaviours of Android apps and profile performance issues, the state-of-the-art methods are limited in their ability to deal with the multiple-layer nature of Android, and thus they cannot analyze and correlate an app's behaviours in various layers and track its cross-layer information leakage flow. In this thesis, we propose novel cross-layer dynamic analysis mechanisms and develop efficient tools to inspect Android apps. More precisely, we propose Malton, a novel on-device non-invasive analysis platform for the new Android runtime, i.e., the ART runtime. Malton runs on real mobile devices and provides a comprehensive view of malware's behaviours by conducting multi-layer monitoring and information flow tracking, as well as efficient path exploration. We have evaluated Malton using real-world malware samples. The experimental results showed that Malton is more

effective than the existing tools, with the capability to analyze sophisticated malware samples and provide a comprehensive view of malicious behaviours of these samples.

Android malware are also becoming more and more sophisticated to evade detection and analysis; one of the most popular techniques adopted by Android malware to protect themselves is packing. Packing services were provided to protect benign apps from being pirated, but Android malware authors also start to adopt packing services to protect the malware from being detected and analyzed. Hence, we propose a novel adaptive approach and develop *PackerGrind*, a new tool based on cross-layer inspection, to unpack Android apps. Packing services (or packers) have been used by not only app developers to protect their apps but also attackers to hide the malicious component and evade the detection. Although there are a few recent studies on unpacking Android apps, it has been shown that the evolving packers can easily circumvent them because they are not adaptive to the changes of packers. *PackerGrind* can reveal the protection mechanisms of packers, recover the Dex files with low overhead, and handle the evolution of packers.

In addition, to improve the efficiency of malware detection, there are various cloud-based Android malware detection approaches proposed. These approaches obtain the behaviors of Android apps on mobile devices, then upload the obtained information to the server for malware detection, and obtain the detection result when detection finishes. When we use these cloud-based detection mechanism, we need take the network performance into account because, if we upload the behaviours of apps and download detection results under poor network status, network traffic jam could be caused. However, when we measure the network performance using existing mobile network measurement apps, there are a set of factors (e.g., Android system architecture and implementation patterns) that could affect the measurement results. We employ the cross-layer dynamic analysis mechanism to conduct the first systematic study on the factors that could bias the measurement results of network features. In particular, we identify new factors,

revisit known factors, and propose a novel approach with new tools to discover these factors in proprietary apps. We also develop a new measurement app named MobiScope for demonstrating how to mitigate the negative effects of these factors and obtain the accurate and stable network features. The extensive experimental results illustrate the negative effects of various factors and the improvement in network measurement brought by MobiScope.

Keywords: Cross-Layer Tracking; Android Profiling; Dynamic Analysis; Unpacking

Publications

1. **Lei Xue**, Yajin Zhou, Ting Chen, Xiapu Luo, and Guofei Gu, “Malton: Towards On-Device Non-Invasive Mobile Malware Analysis for ART”, in *Proceedings of the 26th USENIX Security Symposium (Security’17)*, Vancouver, BC, Canada, August 16-18, 2017.
2. **Lei Xue**, Xiapu Luo, Le Yu, Shuai Wang, and Dinghao Wu, “Adaptive Unpacking of Android Apps”, in *Proceedings of 39th International Conference on Software Engineering (ICSE’17)*, Buenos Aires, Argentina, May 20-28, 2017.
3. **Lei Xue**, Xiaobo Ma, Xiapu Luo, Le Yu, Shuai Wang, and Ting Chen, “Is What You Measure What You Expect? Factors Affecting Smartphone-Based Mobile Network Measurement”, in *Proceedings of IEEE International Conference on Computer Communications (INFOCOM’17)*, Atlanta, GA, USA, May 1-4, 2017.
4. Le Yu, Tao Zhang, Xiapu Luo, **Lei Xue**, and Henry Chang, “Towards Automatically Generating Privacy Policy for Android Apps”, *IEEE Transactions on Information Forensics and Security (TIFS)*, March, 2017.
5. Xiapu Luo, Haochen Zhou, Le Yu, **Lei Xue**, and Yi Xie, “Characterizing mobile *-box applications”. *Computer Networks (COMNET)*, July, 2016.
6. Le Yu, Tao Zhang, Xiapu Luo, and **Lei Xue**, “AutoPPG: Towards Automatic

- Generation of Privacy Policy for Android Applications”. in *Proceedings of 6th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM'15)*, Denver, Colorado, USA, October 12-16, 2015.
7. **Lei Xue**, Chenxiong Qian, and Xiapu Luo, “AndroidPerf: A Cross-layer Dynamic Profiling System for Android”, in *Proceedings of IEEE/ACM International Symposium on Quality of Service (IWQoS'15)*, Portland, OR, USA, June 15-16, 2015.
 8. **Lei Xue**, Xiapu Luo, Edmond W. W. Chan, and Xian Zhan, “Towards Detecting Target Link Flooding Attack”. in *Proceedings of 28th USENIX Large Installation System Administration Conference (LISA'14)*, Seattle, USA, November 9-14, 2014.
 9. Xiapu Luo, **Lei Xue**, Cong Shi, Yuru Shao, Chenxiong Qian, and Edmond W. W. Chan, “On Measuring One-Way Path Metrics from a Web Server”, in *Proceedings of IEEE International Conference on Network Protocols (ICNP'14)*, Research Triangle Park, NC, USA, October 21-24, 2014.
 10. **Lei Xue**, Xiapu Luo, and Yuru Shao, “kTRxer: A portable toolkit for reliable internet probing”, in *Proceedings of IEEE/ACM International Symposium on Quality of Service (IWQoS'14)*, Hong Kong, China, May 26-27, 2014.

Acknowledgements

First and foremost, I am deeply grateful to all the nice and great people I have met or known. Although many of them may not be mentioned, what I learned from them inspires everything I have accomplished.

Regarding the thesis, I would like to especially thank my supervisor, Prof. Xiapu Luo. He always encouraged me to do the research with my own interest and discussed every research questions and ideas with me in detail during my whole study period. I have benefited a lot from his expertise, vast knowledge and skill in different research areas.

There are also many other teachers and classmates that helped me a lot during my study period. I would like to thank Xiaobo Ma, Ting Chen, Chenxu Wang and Tao Zhang for providing novel inspirations and suggestions when I discussed with them, and their rigorous scientific approaches helped me become a well-trained researcher. I would also like to thank all the members of my group. The friendly working environment created by them is essential for my study, and the time I worked and discussed with them is a nice memory in my life.

Finally, and most importantly, I would like to gratefully and sincerely thank my parents, sisters, and relatives for their unyielding support and encouragement, which allow me to follow my dream and finish this thesis.

Contents

Abstract	i
Publications	v
Acknowledgements	vii
List of Figures	xiii
List of Tables	xv
1 Introduction	1
1.1 Static Analysis	1
1.2 Dynamic Analysis	2
1.3 Our Work	4
1.3.1 Android Malware Analysis	5
1.3.2 Android Unpacking	6
1.3.3 Android App Profiling	8
1.4 Thesis Outline	9
2 Literature Review	11
2.1 Static Analysis	11
2.2 Android Malware Analysis	13
2.2.1 Dynamic and Hybrid Analysis	13
2.2.2 Multi-path Analysis for Android	17

2.3	Android Unpacking	18
2.4	Android App Profiling	20
3	On-Device Non-Invasive Mobile Malware Analysis for ART	23
3.1	Overview	23
3.2	Background	24
3.2.1	The ART Runtime	24
3.2.2	A Motivating Example	25
3.3	Malton	27
3.3.1	Overview	28
3.3.2	Android Framework Layer	29
3.3.3	Android Runtime Layer	33
3.3.4	System Layer	34
3.3.5	Instruction Layer: <i>Taint Propagation</i>	36
3.3.6	Instruction Layer: <i>Path Exploration</i>	37
3.4	Evaluation	41
3.4.1	Sensitive Behavior Monitoring	41
3.4.2	Malware Analysis	43
3.4.3	Path Exploration	50
3.4.4	Performance Overhead	53
4	Adaptive Unpacking of Android Apps	55
4.1	Overview	55
4.2	Background	56
4.2.1	Dex File	56
4.2.2	Android App Packing	57
4.2.3	A Motivating Example	58

4.3	Basic Dex Data Collection Points	60
4.3.1	Dalvik VM (DVM)	60
4.3.2	Android Runtime (ART)	61
4.4	PackerGrind	63
4.4.1	Overview	63
4.4.2	Monitoring	64
4.4.3	Tracking report	67
4.4.4	Recovery	67
4.4.5	Analysis	70
4.4.6	Implementation on ART	73
4.5	Evaluation	75
4.5.1	Data Set	75
4.5.2	Protection Mechanisms	76
4.5.3	Recovering Dex Files	77
4.5.4	Comparison	80
4.5.5	Unpacking Malware	82
4.5.6	Overhead	84
5	Scrutinizing Smartphone-Based Mobile Network Measurement Apps	87
5.1	Overview	87
5.2	Background	88
5.2.1	Mobile Network Measurement	88
5.2.2	A Motivating Example	89
5.3	Factors Affecting Measurement Results	90
5.3.1	Category 1: Implementation Patterns	92
5.3.2	Category 2: Android Architecture and Configurations	94

5.3.3	Category 3: Network Protocols	95
5.4	Is an App Affected by These Factors?	96
5.4.1	Static Bytecode Analysis	97
5.4.2	AppTracer: Dynamic Trace Analysis	99
5.5	MobiScope	102
5.6	Evaluation	103
5.6.1	Testbed	103
5.6.2	Effect of Factors in Category 1	104
5.6.3	Effect of Factors in Category 2	106
5.6.4	Effect of Factors in Category 3	111
5.6.5	Evaluation of kping and kband	113
5.7	Examining Descriptions of Measurement Apps	114
5.7.1	Questionnaire Design	115
5.7.2	Result Analysis	115
6	Conclusion	117
6.1	Conclusion	117
6.2	Future Work	118
	Bibliography	120

List of Figures

3.1	The scenario of Malton.	28
3.2	The overview of the system implementation.	30
3.3	The example to parse the result of <i>TelephonyManager.getDeviceId()</i>	31
3.4	The detailed flow of contact related information tracking of the <i>XXshenqi</i> malware (Grey diagrams represent taint sources, diagrams with bold lines represent taint sinks, rectangles represent data, ellipses represent behaviours (methods) and red italics strings represent tainted information).	44
3.5	Detect stealthy behaviours through the Java/JNI reflection of the <i>photo3</i> malware (ellipse represents behaviours of the Android framework behaviour, and rectangle represents the behaviours of the system layer).	46
3.6	The major information collected by Malton at function level. The names of Android runtime functions and system calls are in black italics. We omit the information of method arguments due to the limited space.	48
3.7	The behaviours reconstructed by Malton.	49
3.8	Performance measured by CF-Bench.	53
4.1	<i>onCreate()</i> method of the app packed by the Baidu packer of DB-15.	59
4.2	The process from Dex file loading to method execution.	60
4.3	The iterative process realized by <i>PackerGrind</i>	63
4.4	Architecture of <i>PackerGrind</i>	64
4.5	Tracking report for an app packed by the Ali packer.	68
4.6	The method <i>onCreate()</i> before and after packing and the tracking report of <i>A.V()</i>	72

4.7	Content of <i>onCreate001()</i> after the 2nd run.	78
4.8	CF-Benchmark results.	84
5.1	An example of RTT measurement conducted on an Android smartphone. We use dash-dot line to connect t_2 and t_{2-k} because the TCP SYN packet is sent by the kernel but triggered by the function at the runtime layer. Similarly, the dash-dot line connecting t_{3-k} and t_3 indicates that the TCP SYN/ACK packet is received by the kernel without forwarding to the runtime layer but the system will notify the function at that layer.	91
5.2	Implementation patterns for HTTP-based and TCP-based RTT measurement.	93
5.3	A snippet of the static and dynamic analysis results of HTTPing.	97
5.4	Architecture of AppTracer	100
5.5	Example of reconstructing invocation relationship from tracing result. . .	101
5.6	The testbed.	103
5.7	HTTP/TCP-based RTT measurement by different ping apps.	104
5.8	Time consumed for sending a UDP packet across different layers.	107
5.9	RTT measured with and without <i>VPNservice</i>	110
5.10	RTT measured with different intervals (Cellular).	110
5.11	RTT measured with different intervals (WiFi).	110
5.12	RTT measured with different added delays (WiFi).	110
5.13	Evaluation of <i>kping</i> and <i>kband</i>	114
5.14	A snippet of the questionnaire.	115

List of Tables

2.1	Comparison of Malton with the existing popular Android malware analysis tools.	14
3.1	Runtime behaviours related functions.	33
3.2	The taint propagation related IR Statements.	35
3.3	Taint propagation related IR expressions.	36
3.4	Comparison on the capability of capturing the sensitive behaviours of malware samples.	43
3.5	The commands and related behaviours explored by Malton (The 3rd column represents the number of IR blocks, required to execute for exploring the behaviour of the corresponding command, with/without in-memory optimization).	51
3.6	The number of IR blocks executed for path exploration with and without in-memory optimization.	52
4.1	Wrappers for tracking Dex and DVM related events.	65
4.2	Wrappers for tracking Dex and ART related events.	74
4.3	Protection mechanisms adopted by six packers in DB-15 and DB-16. The symbol before (or after) “—” denotes whether a packer in DB-15 (or DB-16) uses the mechanism or not.	76
4.4	Number of runs required for determining all Dex data collection points.	77
4.5	Difference between the original Dex file and recovered Dex file from the samples of DB-15/DB-16 (\oplus , \ominus and \odot represent the recovered Dex file has additional code, less code, and the same code compared with the original Dex file, respectively).	79
4.6	Comparison among Android-unpacker, DexHunter and <i>PackerGrind</i>	81

4.7	Permissions and sensitive API calls in malware samples before and after unpacking by <i>PackerGrind</i>	83
5.1	The implementation patterns of 10 measurement apps under examination. <code>ping</code> refers to using the default <code>ping</code> tool in Android.	105
5.2	Packet transmission capacity measured with different configurations (normal/with <code>libpcap</code> /with 10 Netfilter rules).	109

Chapter 1

Introduction

Android has become the most popular mobile operating system platform and billions of Android applications (apps) have been downloaded from the official and third-party markets. Understanding the behaviours of Android apps is essential for identifying their malicious actions and locating their potential performance issues. For program analysis, there are two common methodologies, static analysis [68, 167] and dynamic analysis [118, 147].

1.1 Static Analysis

Static analysis on Android privacy leakage mainly focuses on permission and bytecode without executing the apps [94, 198, 93]. For example, *Stowaway* [86] assigns detailed permission to each API call and IPC Inspection [87] detects attacks through Android permission system. These studies usually first use reverse engineering tools, such as *smali* [102] and *ded* [83] to disassemble the APKs, and then conduct the analysis and detection [92, 196, 197]. However, various protection techniques have been applied to impede static analysis [123], such as packing, reflection and dynamic loading. Moreover, since static analysis cannot obtain the dynamic behaviours of the Android apps, it

is difficult for them to handle the dynamic features of programming languages and locate potential performance issues. For instance, malware exploits various obfuscation techniques to raise the bar of code comprehension [141], implements malicious activities in native libraries to evade the inspection [197, 40, 159, 137], and leverages packing techniques to hide malicious payloads [193, 182, 178].

1.2 Dynamic Analysis

A number of dynamic analysis approaches for Android apps have been proposed with different purposes, such as malware analysis [82, 161, 192, 180] and Android performance analysis [144, 145, 185]. However, they have several limitations in inspecting the dynamic behaviors of apps. First, the majority of dynamic analysis [163, 79, 195, 105] lack of the capability of cross-layer inspection, and thus provide incomplete view of apps' behaviours. For example, CopperDroid [163] monitors apps' behaviours mainly through the trace of system calls (e.g., *sys_sendto()* and *sys_write()*). Thus, it is hard to expose the execution details in the Android framework layer and the runtime layer, due to the well-known semantic gap challenge.

Second, most of the existing tools rely on emulators (e.g., DroidScope [180]) or debugging techniques (e.g., ptrace) or modifying the old Android runtime (i.e., Dalvik Virtual Machine, or DVM for short) to monitor malware behaviours (e.g., TaintDroid [82]). Therefore, they become ineffective if the apps adopt anti-debug and anti-emulator techniques [134, 166, 103, 108]. For example, in [43], 98.6% malware samples were successfully analyzed on the real smartphone, whereas only 76.84% malware samples were successfully inspected using the emulator. Moreover, the new Android runtime (i.e., the ART runtime) further limits the usage of some dynamic analysis tools.

Since mobile devices usually have limited resources, there are various cloud-based Android malware detection approaches proposed for improving the malware detection efficiency [98, 199, 125, 187]. Although many measurement apps have been developed and published in Google Play or Apple Store [14] for measuring the network performance, the measurement results from apps may be not what users have expected because of two reasons. Various factors could affect the measurement results, and not all app developers are network measurement experts who are aware of such factors. For example, according to RFC 2681 [42], the round-trip time (RTT) reported by an app is determined by the *host times*, which include the timestamp just prior to sending the packet and that right after receiving the response packet. In contrast, the network RTT is calculated using the *wire times*, which refer to the time when the packet leaves the smartphone's network interface (NIC) and the time when the corresponding response packet arrives at the smartphone's NIC. It has been shown that the difference between *host times* and *wire times* cannot be neglected [114]. Users could misunderstand the results from the mobile measurement apps because their descriptions may be ambiguous. For example, although many apps claim to be able to measure *RTT*, they refer to different time intervals, such as RTTs derived from host times/wire times and time-to-first-byte with/without DNS resolution.

Besides profiling apps' behaviors, dynamic analysis has been widely employed in many applications, such as unpacking hardened apps [193, 182], locating performance issues [114, 148, 96], etc. Unfortunately, the shortcomings in existing dynamic analysis approaches limit their usages in these applications. For example, attackers utilize packers to hide malware for evading the signature-based detection and impeding the investigation of their malicious behaviours [47]. Although a few unpacking approaches have been recently proposed to recover the Dex files from packed apps [193, 182], the latest version of packers could easily evade those unpacking tools. The key issue lies in the one-pass processing strategy adopted by the unpacking tools. We address this issue by proposing a

novel adaptive unpacking approach based on our cross-layer dynamic analysis technique.

Knowing the performance of mobile network is important to many applications [21, 91]. For example, since mobile devices have limited computational resources, various cloud-based malware detection systems have been proposed for improving the detection efficiency. Such systems analyze the apps from the smartphone and send back the analysis result. To quickly submit an app to the cloud-based analysis system, apps rely on network measurement result to select the proper cloud server. Although many measurement apps have been developed and published in Google Play or Apple Store [14], the measurement results from apps may not be what users have expected because various factors could affect the measurement results and not all app developers are network measurement experts who are aware of such factors. Although a few recent studies pointed out some factors (e.g., Dalvik virtual machine) that may affect the measurement [114, 148, 96], they have several limitations, such as missing other important factors (e.g., implementation patterns), conducting only coarse-grained analysis, and lack of evaluating off-the-shelf measurement apps. We propose a new methodology based on our cross-layer dynamic analysis technique to reveal the factors that affect the measurement results of such apps.

1.3 Our Work

We concentrate on designing and developing effective and efficient approaches to facilitate dynamic behavior analysis of Android apps. Moreover, we apply our new techniques to addressing several challenging problems, including cross-layer malware behavior profiling, adaptive unpacking of hardened apps, and locating the factors that impact the network performance measured by apps.

1.3.1 Android Malware Analysis

First, we propose Malton, a novel on-device non-invasive analysis platform for the new Android runtime, i.e., the ART runtime. Malton runs on real mobile devices and provides a comprehensive view of malware’s behaviours by conducting multi-layer monitoring and information flow tracking, as well as efficient path exploration. Second, we propose a novel adaptive approach and develop *PackerGrind*, a new tool based on cross-layer inspection to unpack Android apps. Finally, we conduct the first systematic study on the factors that could bias the result from measurement apps and their descriptions by leveraging the cross-layer analysis. In particular, we identify new factors, revisit known factors, and propose a novel approach with new tools to discover these factors in proprietary apps.

In this thesis, we propose Malton, a novel on-device non-invasive analysis platform for the ART runtime. Compared with other systems, Malton has two important capabilities, namely, a) multi-layer monitoring and information flow tracking, and b) efficient path exploration, which provide a comprehensive view of malware behaviours. Moreover, Malton does not need to modify malware’s bytecode for conducting static instrumentation. To our best knowledge, Malton is the *first* system with such capabilities. Table 2.1 in Chapter 2.2.1 illustrates the key differences between Malton and other systems.

Specifically, Malton inspects Android malware on different layers. It records the invocations of Java methods, including sensitive framework APIs and the concerned methods of the malware, in the *framework layer*, and captures stealthy behaviours, such as dynamic code loading and JNI reflection, in the *runtime layer*. Moreover, it monitors library APIs and system calls in the *system layer*, and propagates taint tags and explores different code paths in the *instruction layer*. However, multi-layer monitoring is not enough to provide a comprehensive view of malware behaviours, because malicious

payloads could be conditionally executed. We deal with this challenge with the capability to efficiently explore code paths. First, to trigger as many malicious payloads as possible, we propose a multi-path exploration engine based on the concolic execution [66] to generate concrete inputs for exploring different code paths. Second, to conduct efficient path exploration on mobile devices with limited computational resources, we propose an offloading mechanism to move heavy-weight tasks (e.g., solving constraints) to resourceful desktop computers, and an in-memory optimization mechanism that makes the execution flow return to the entry point of the interested code region immediately after exiting the code region. Third, in case that the constraint solver fails to find a solution to explore a code path, we equip Malton with a direct execution engine to forcibly execute a specified code path. Since Malton requires the necessary human annotations of the interested code regions, it is most useful in the human-guided detailed exploration of Android malware.

We have implemented a prototype of Malton based on the binary instrumentation framework Valgrind [126]. Since both the app's code and the framework APIs are compiled into the native code in the ART runtime, we leverage the instrumentation mechanism of Valgrind to introspect apps and the Android framework. We evaluated Malton with real-world malware samples. The experimental results showed that Malton is able to analyze sophisticated malware samples, and provides a comprehensive view of their malicious behaviours.

1.3.2 Android Unpacking

In this thesis, we also apply our methodology to solve other challenging issues that require careful cross-layer analysis. We first propose a new adaptive approach, which employs an iterative process, to recover the Dex files from packed apps, and develop a new system

named *PackerGrind* to automate most steps in the process.

Our iterative process consists of three major tasks including (1) *monitoring*, which captures how packed apps work, especially how it prepares the real code for execution, and then generates tracking reports, based on which we can determine the data collection points; (2) *recovery*, which collects the pieces of data in Dex files at the selected data collection points and reconstructs Dex files; (3) *analysis*, which determines whether new data collection points are needed to recover Dex files. Automating this process is not trivial because we need to address two challenges:

- How to conduct cross-layer profiling of packed apps' behaviours in a smartphone?
- How to effectively recover the Dex files of apps packed by different packers?

Resolving the first challenge needs a system that can perform cross-layer monitoring of an app's behaviours and run in real smartphones. Note that with the support of Android framework, apps run in the runtime, which was the Dalvik Virtual Machine (DVM) before Android 5.0 and became the new Android runtime (ART) afterward, and the runtime is on top of the modified Linux. Packed apps usually exploit the features of the Java language, Android framework, and native libs/instructions to hide the real code, detect emulator, and prohibit debugging [166]. Existing dynamic analysis systems for monitoring apps cannot address the first challenge, because they either rely on emulator (e.g., Qemu) [180, 163] and debugging techniques [195, 181] or lack of the support of cross-layer profiling [82, 163]. To tackle the second challenge, we propose and develop a novel cross-layer monitoring component for *PackerGrind*. By exploiting dynamic binary translation [126], it collects information from the runtime, the system, and the instruction layers and runs in smartphones. Moreover, it supports both DVM and ART.

Existing unpackers for Android apps cannot fully address the second challenge because

of their one-pass processing strategy. To address the second challenge, we first identify basic Dex data collection points by scrutinizing how DVM and ART load and run apps. Since different packers employ various protection methods to modify the code and data in memory dynamically, *PackerGrind* provides detailed tracking reports as well as suggested criteria to recognize the protection patterns. Moreover, *PackerGrind* conducts static analysis on the Dex file obtained at each run to facilitate users to identify new data collection points if needed. Although this step might need manual inspection, the detailed information and scripts provided by *PackerGrind* could alleviate the workload. *PackerGrind* also has built-in rules to automatically unpack apps protected by existing packers accessible to us. After that, *PackerGrind* will re-run the packed app, collect Dex data at selected collection points, and finally reconstruct the Dex file.

1.3.3 Android App Profiling

In addition, we conduct the *first* systematic study of the factors that could bias the result from measurement apps and their descriptions. It is challenging to accomplish this study because the measurement process involves intricate factors from apps, OS, and network protocols. Moreover, it is difficult and time-consuming to understand how an app performs the measurement, not to mention that most apps are proprietary.

We examine Android system, apps, and network protocols to identify new factors and revisit known factors, and perform extensive experiments to quantify their effect. Android system is selected because it has occupied more than 81% market share [72]. To discover these factors in measurement apps, we develop two tools, namely *AppDissector*, a static bytecode analyzer, and *AppTracer*, a dynamic trace analyzer. We also design *MobiScope*, a measurement app for demonstrating how to mitigate the negative effects of various factors.

Furthermore, we construct enhanced descriptions for measurement apps to provide users more information about what is measured by leveraging the static and dynamic analysis of measurement apps. User studies have been performed to assess whether the original and the enhanced descriptions make users understand what the apps measure.

1.4 Thesis Outline

The rest of this thesis is organized as follows. Chapter 2 introduces the related work. Chapter 3 proposes a novel on-device non-invasive analysis platform for the ART runtime. This platform can provide users a comprehensive view of the Apps' behaviours in different layers. Chapter 4 proposes an adaptive approach to unpack the packed Android apps, and an unpacking tool is implemented based on this approach to unpack the packed malware adaptively. Chapter 5 conducts the *first* systematic study of the factors that could bias the result from measurement apps and their descriptions. A measurement app is also implemented to demonstrate how to mitigate the negative effects of various factors in this chapter. Finally, Chapter 6 makes a conclusion of this thesis and indicates future work.

The primary research outputs emerged from this thesis are as follows:

- Lei Xue, Yajin Zhou, Ting Chen, Xiapu Luo, and Guofei Gu, "Malton: Towards On-Device Non-Invasive Mobile Malware Analysis for ART", in *Proceedings of the 26th USENIX Security Symposium (Security'17)*, Vancouver, BC, Canada, August 16-18, 2017.
- Lei Xue, Xiapu Luo, Le Yu, Shuai Wang, and Dinghao Wu, "Adaptive Unpacking of Android Apps", in *Proceedings of 39th International Conference on Software Engineering (ICSE'17)*, Buenos Aires, Argentina, May 20-28, 2017.
- Lei Xue, Xiaobo Ma, Xiapu Luo, Le Yu, Shuai Wang, and Ting Chen, "Is What

You Measure What You Expect? Factors Affecting Smartphone-Based Mobile Network Measurement”, in *Proceeding of IEEE International Conference on Computer Communications (INFOCOM’17)*, Atlanta, GA, USA, May 1-4, 2017.

- Lei Xue, Chenxiong Qian, and Xiapu Luo, “AndroidPerf: A Cross-layer Dynamic Profiling System for Android”, in *Proceedings of IEEE/ACM International Symposium on Quality of Service (IWQoS’15)*, Portland, OR, USA, June 15-16, 2015.
- Lei Xue, Xiapu Luo, and Yuru Shao, “kTRxer: A Portable Toolkit for Reliable Internet Probing”, in *Proceedings of IEEE/ACM International Symposium on Quality of Service (IWQoS’14)*, Hong Kong, China, May 26-27, 2014.

Chapter 2

Literature Review

Being one of the most popular mobile operating systems, Android has occupied 81.7% market share [33] and owned more than 2.9 million application (or simply apps) in Google Play market [34]. Therefore, various approaches have been proposed to analyze Android apps for either identifying potential performance issues or detecting malicious behaviours. These analysis approaches can be divided into two major types, static analysis [68, 167] and dynamic analysis [118, 147].

2.1 Static Analysis

Static analysis has been widely used for malware detection and security vulnerabilities finding for a long time, such as *Stowaway* [86] that assigns detailed permission to each API call and *IPC Inspection* [87] that detects attacks through Android permission system. Most of the static analysis techniques are suitable for Java code and APK analysis. They disassemble the APKs and reveal the malicious codes by reverse engineering tools like *smali* [102] and *ded* [83]. Then the malicious behaviours are detected based on the disassembled codes [92, 196, 197]. There are already many similar static analysis

systems [94, 198, 93].

FlowDroid [48] is proposed to perform a highly precise taint flow static analysis for each component in an Android application, and it builds the call graph based on Spark/Soot [16], which conducts a flow insensitive points to analysis. IccTA [112] also adopts the static taint analysis approach to track information leakage through inter-component communication. IntelliDroid [172] is introduced as a generic Android input generator that helps the dynamic analysis tools quickly identify and analyze malicious behaviours through generating inputs specific to the dynamic analysis tools. The open-source reverse engineering tool Androguard [78] can perform various operations on the APK files and identify malicious and benign applications based on the static signatures. In [57], a static analysis method is presented to specifically track two types of implicit control flow, Java reflections and Android intents, which are frequently used in Android apps. DroidNative [40] detects malware in either bytecode or native code through static analysis of the native code and focuses on patterns in the control flow that are not significantly impacted by obfuscations. AppIntent [183] can efficiently provide a sequence of GUI manipulation related to the events leading to the data transmission, and it focuses on determining whether the data transmission is user intended or not. SmartGen [200] is proposed to expose the hidden malicious URLs to assist Android malware analysis through selective symbolic execution.

However, none of these static analysis systems can obtain the dynamic data flow and reconstruct runtime system view. Various effective protection techniques have been applied to evade static analysis [123], such as packing, reflection and dynamic loading. Moreover, static analysis techniques cannot obtain the dynamic behaviours of the Android apps during their execution, hence these techniques do not adapt to analyzing the performance issues of the Android apps.

2.2 Android Malware Analysis

Currently, various dynamic approaches have been proposed to profile apps [144, 145, 185] and monitor the behaviours of apps [82, 161, 192, 180]. However, these approaches are limited in their ability to deal with the multiple-layer nature of Android, and thus cannot uncover issues due to the underlying platform or poor interactions between different layers, and identify the malicious behaviours in various layers.

Android malware analysis techniques can be generally divided into static analysis, dynamic analysis, and the hybrid of static and dynamic analysis. Since Malton is a dynamic analysis system, this section introduces the involved with related dynamic and hybrid analysis. Interested readers please refer to [149, 162, 175] for more information on static analysis of Android apps.

2.2.1 Dynamic and Hybrid Analysis

According to the implementation techniques, the existing (dynamic or hybrid) Android malware analysis tools can be roughly divided into five major types: 1) tailoring Android system [82, 79, 192, 161], 2) customizing Android emulator Qemu [180, 163], 3) modifying (repackaging) app implementation [81], 4) employing system tracking tools [181], or 5) restricting by an app sandbox [52, 59].





We compare Malton with existing popular (dynamic or hybrid) Android malware analysis tools, and show the major differences in Table 2.1. Please note that , , and  indicate that the tool can capture malware behaviours in the framework layer, the runtime layer and the native layer, respectively. Besides, the shadow sector means partial support. For example,  of TaintART suggests that it can monitor partial framework behaviours.

Table 2.1: Comparison of Malton with the existing popular Android malware analysis tools.

Tool	On device	Non-invasive	Support ART	Cross-layer Monitoring	Multi-path analysis	In-memory mechanism	Offload mechanism	Direct execution	Without modifying OS	Type
TaintDroid [82]	✓	✓	×	●	×	×	×	×	×	Dynamic
TaintART [161]	✓	×	✓	⊗	×	×	×	×	×	Dynamic
ARTist [53]	✓	×	✓	⊗	×	×	×	×	×	Dynamic
DroidBox [79]	✓	✓	×	●	×	×	×	×	×	Dynamic
VeDroid [192]	✓	✓	×	●	×	×	×	×	×	Dynamic
DroidScope [180]	×	✓	×	●	×	×	×	×	✓	Dynamic
CoppertDroid [163]	×	✓	✓	●	×	×	×	×	✓	Dynamic
Dagger [181]	✓	✓	✓	⊗	×	×	×	×	✓	Dynamic
ARTDroid [73]	✓	✓	✓	⊗	×	×	×	×	✓	Dynamic
Boxify [52]	✓	✓	✓	●	×	×	×	×	✓	Dynamic
CRPE [71]	✓	✓	×	⊗	×	×	×	×	×	Dynamic
DroidTrace [195]	✓	✓	✓	⊗	×	×	×	×	✓	Dynamic
DroidTrack [39]	✓	✓	×	⊗	×	×	×	×	×	Dynamic
MADAM [80]	✓	✓	✓	●	×	×	×	×	×	Dynamic
HARVESTER[141]	✓	✓	✓	⊗	✓	×	×	✓	✓	Hybrid
AppAudit[174]	×	×	×	⊗	×	×	×	×	×	Hybrid
GroddDroid[35]	✓	×	✓	⊗	✓	×	×	✓	✓	Hybrid
ProfileDroid [169]	✓	✓	✓	●	×	×	×	×	✓	Hybrid
Malton	✓	✓	✓	●	✓	✓	✓	✓	✓	Dynamic

TaintDroid [82] conducts dynamic taint analysis to detect information leakage by modifying DVM. It does not capture the behaviours in native layer because it trusts the native libraries loaded from firmware and does not consider third-party native libraries. While only a small percent of apps used native libraries when TaintDroid was designed, recent studies showed that native libraries have been intensively used by apps and malware [49, 137]. At the runtime layer, although TaintDroid can track taint propagation in DVM, it neither monitors the runtime behaviours nor supports ART. Many studies [79, 192, 194, 142, 150, 170, 155] have enhanced TaintDroid from different aspects, but they do not achieve the same capability as Malton does. For example, AppsPlayground [142] combines TaintDroid and fuzzing to conduct multi-path taint analysis. Mobile-Sandbox [155] uses TaintDroid to monitor framework behaviours and employs ltrace [13] to capture native behaviours.

To avoid modifying Android system (including the framework, native libraries, Linux kernel etc.), a number of studies [141, 191, 37, 153, 54, 176, 58, 173, 140, 74, 35, 75, 101] propose inserting the logics of monitoring behaviours or security policies into the Dalvik bytecode of the malware under inspection and then repacking it into a new APK. Those studies have three drawbacks in common. First, they can only monitor the framework layer behaviours by manipulating Dalvik bytecode. Second, those approaches are invasive that can be detected by malware. Third, malware may use packing techniques to prevent such approaches from being repackaged [182, 193].

Based on Qemu, DroidScope [180] reconstructs the OS-level and Java-level semantics, and exports APIs for building specific analysis tools, such as dynamic information tracer. Hence, there is a semantic gap between the VMI observations and the reconstructed Android-specific behaviours. Since it monitors the Java-level behaviours by tracing the execution of Dalvik instructions, it cannot monitor the Java methods that are compiled into native code and running on ART (i.e, partial support of framework layer). Moreover,

DroidScope does not monitor JNI and therefore it cannot capture the complete behaviours at runtime layer. CopperDroid [163] is also built on top of Qemu and records system call invocations by instrumenting Qemu. Since it performs binder analysis to reconstruct the high-level Android-specific behaviours, only a limited number of behaviours can be monitored. In addition, it cannot identify the invocations of framework methods. ANDRUBIS [117] and MARVIN [116] (which is built on top of ANDRUBIS) monitor the behaviours of framework layer by instrumenting DVM, and logging system calls of native code by VMI.

Monitoring system calls [181, 168, 130, 195, 105, 80, 155, 115, 46, 169] is widely used in Android malware analysis because a considerable amount of APIs in upper layers are eventually realized by systems calls. For instance, Dagger [181] collects system call information through strace [17], recodes binder transactions via sysfs [18], and accesses process details from */proc* file system. One common drawback of system-call-based techniques is the semantic gap between system calls with the behaviours of upper layers, even though several studies [181, 130, 195] try to reconstruct high-level semantics from system APIs. Besides tracing system calls, MADAM [80] and ProfileDroid [169] monitor the interactions between user and smartphone. However, they cannot capture the behaviours in runtime layer.

Both TaintART [161] and ARTist [53] are new frameworks to propagate the taint information in ART. They modify the compiler `dex2oat`, which is provided along with ART runtime to turn Dalvik bytecode into native code during app's installation, so that the taint propagation instructions will be inserted into the compiled code by the modified `dex2oat`. They only propagate taint at runtime layer, and do not support taint propagation through JNI or in native code. Moreover, they cannot handle the packed malware, because such malware usually dynamically load the Dalvik bytecode into runtime directly without triggering the invocation of `dex2oat`. CRePE [71] and

DroidTrack [39] track apps' behaviours at the framework layer by modifying Android framework.

Boxify [52] and NJAS [59] are app sandboxes that encapsulate untrusted apps in a restricted execution environment within the context of another trusted sandbox application. Since they behave as the proxy for all system calls and binder channels of isolated apps, they support the analysis of native code and could reconstruct partial framework layer behaviours.

ARTDroid [73] traces framework methods by hooking the virtual framework methods and supports ART. Since the boot image boot.art contains both the *vtable_* and *virtual_methods_* arrays that store the pointers to virtual methods, ARTDroid hijacks *vtable_* and *virtual_methods_* to monitor the APIs invoked by malware.

HARVESTER and GroddDroid [141, 35] support multi-path analysis. The former [141] covers interested code forcibly by replacing conditionals with simple Boolean variables, while the latter [35] uses a similar method to jump to interested code by replacing conditional jumps with unconditional jumps. Different from Malton, they need to modify the bytecode of malware.

2.2.2 Multi-path Analysis for Android

There are a few studies about multi-path analysis for Android. TriggerScope [88] is a static symbolic executor that handles Dalvik bytecode. Similar to other static analysis tools, it may run into trouble when handling reflections, native code, dynamic Dex loading etc.. Anand et al. [44] proposed ACTEve that uses concolic execution to generate input events for testing apps and offloaded constraint solving to the host. There are three major differences between ACTEve and the path explorer of Malton. First, since ACTEve

instruments the analyzed app and the SDK, this invasive approach may be detected by malware. Second, ACTEve does not support native code. Third, it does not apply the in-memory optimization. ConDroid [152] also depends on static instrumentation, and therefore has the same limitations as ACTEve.

Two recent studies [183, 120] propose converting Dalvik bytecode into Java bytecode and then using Java PathFinder [45] to conduct symbolic execution in a custom-made JVM. However, JVM cannot properly emulate the real device. Also, SmartGen [200] conducts symbolic execution based on the ECG (Extended Control Graph) which is built on the static analysis of APK files. Consequently, all these tools do not support the analysis of native code.

To make concolic execution applicable for testing embedded software, Chen et al.'s work [65] and MAYHEM [62] adopt similar offloading approach. However, they do not apply the in-memory optimization and cannot be used to analyze Android malware. For example, Chen et al.'s work coordinates the part on device and the part on host through the Wind River Tool Exchange protocol which is designed especially for VxWorks. EvoDroid [119] utilizes the evolutionary algorithm to generate test inputs automatically. However, it focuses on UI testing and relies on static analysis to identify UI elements and the call graph.

2.3 Android Unpacking

Although there are already many studies on code packing/unpacking, most of them focus on x86 native codes [95, 60, 165, 151]. The unpacking techniques for x86 binaries cannot be applied to packed Android apps because Android and the OSes running on x86 CPU have different architectures and execution models [193, 182], let alone the

different formats of their executables. For example, Android packers need to protect both the Dex code and the native code if any, whereas traditional packers only hide native code [95, 60, 165, 151].

Since mobile malware adopts packers to evade the detection, a few studies on unpacking apps were proposed recently from both the academia [193, 182, 107] and the industry [32, 2]. However, all of them adopt the one-pass strategy (i.e., dump the Dex data at fixed points), and therefore they can be easily evaded by the latest packers. For example, DWroidDump [107] only collects the Dex data in *dvmDexFileOpenFromFd()* when a Dex file is mapped to memory by the runtime. DexHunter [193] and AppSpear [182] are proposed to be a general unpacker by customizing Android runtime. DexHunter inserts code in *defineClassNative()* to extract Dex files from memory. However, it may dump invalid Dex files since packers can release the real code after this function. AppSpear instruments the Dalvik interpreter to collect required data during method execution and then reconstructs the Dex file. Unfortunately, AppSpear may also dump invalid data because it relies on DVM's parsing methods to collect Dex data. Note that packers could make these methods return inaccurate results and use their own functions to parse Dex files. Moreover, AppSpear does not support ART. In contrast, *PackerGrind* adopts an iterative approach and conducts cross-layer monitoring so that it is adaptive to the changes of packers. Experimental results show that it outperforms existing approaches. Also, it supports both DVM and ART.

Existing cross-layer monitoring tools [169, 82, 180, 137] for Android cannot collect all necessary information and fulfill the requirement for handling packed apps. For example, ProfileDroid [169] cannot handle packed apps because it relies on apktool to conduct static analysis. TaintDroid [82] neither supports ART nor collects information at the runtime, system, and instruction layers. DroidScope [180] and NDroid [137] rely on Qemu, which can be detected by packers [103].

2.4 Android App Profiling

A number of systems have been designed to profile Android apps for different purposes, such as identifying performance issues [15, 169], detecting malware [82, 142], modeling energy consumption [185, 131], etc.. However, none of them can conduct cross-layer analysis.

Google offers Traceview and dmtracedump to trace invoked functions and collect time spent in each function [15]. Although the trace logs can be generated by including the *Debug* class in an app or using DDMS, Google recommends to use the former to get more precise results [15].

While some systems instrument apps and/or the system to locate performance issues, they neither trace invoked functions nor collect information from the system and the kernel, and thus they cannot reveal issues due to the underlying platform [189, 144, 145, 143]. For example, to measure user-perceived transaction, Panappticon instruments event handlers, asynchronous call interfaces, and the interprocess communication mechanisms to log events [189]. Similarly, AppInsight instruments apps for Windows Mobile to identify the critical execution path for locating performance bottlenecks [144, 145, 143].

Some systems collect information from different sources to profile apps or model energy consumptions [169, 138, 64, 185, 111, 131]. However, they neither track all function calls at different layers nor perform the dynamic taint analysis, and thus cannot uncover issues due to the poor interactions between different layers. For example, ProfileDroid collects user-generated events through adb, system calls through strace, and network traffic through tcpdump to profile apps [169]. ARO characterizes resource usage by correlating user input events and network traffic [138]. QoE Doctor further correlates user interaction events, network traffic, and RRC/RLC layer data through QxDM to diagnose apps QoE [64]. To

estimate energy consumption, AppScope monitors an app’s hardware usage by probing system calls relevant to hardware operations at the kernel level [185, 111]. Being a fine-grained energy profiler, Eprof collects DVM level function calls and system calls by modifying Android framework [131]. If an app has native components, Eprof requires to link it with the Android gprof library [131].

Although VARI profiler can trace invoked functions in the DVM layer, system layer, and kernel layer, it does not perform dynamic taint analysis and thus cannot track information flow across different layers[157].

Many apps for mobile network measurement have been proposed [109, 128, 171, 91]. However, most of them conduct the measurement *without* considering the affecting factors. Although a few studies pointed out some factors, there is a lack of a systematic investigation on them. For example, studies on the effect of mobile network protocols and WiFi on performance [99, 84, 77, 96, 158] neither examine the effect on measurement apps nor take into account apps’ implementation patterns and Android architecture and configurations. Note that all factors under our investigation have non-negligible impact on the result of mobile network measurement.

The most closely related work is [114]. Different from ours, they only studied the effect of DVM [114] and conducted coarse-grained analysis. For example, they attribute the additional delay of HTTP-based RTT measurement to DVM and kernel, neglecting apps’ implementation patterns. Moreover, their analysis has two limitations that may bias their results. First, they monitor packets using *TcpDump*, which will introduce obvious delay as shown in Table 5.2. Second, they use the timestamps of wireless frames to approximate the time when the request (or response) packet leaves (or reaches) the smartphone without considering the contention of WiFi channel, which will also bring additional delay.

Recent studies on profiling app performance [64, 189, 144] aim at improving user

experience rather than mobile network measurement. QoE Doctor [64] diagnoses the user-perceived latency of Android apps by correlating user interaction events, network traffic and RRC states. Panappticon [189] identifies the critical paths in user transactions, and locates performance problems by capturing specified events at the user/kernel layers. AppInsight locates performance bottlenecks through critical paths in user transactions. However, it focuses on Windows mobile apps, and needs to modify the binaries [144].

Chapter 3

On-Device Non-Invasive Mobile Malware Analysis for ART

3.1 Overview

It's an essential step to understand malware's behaviours for developing effective solutions. Though a number of systems have been proposed to analyze Android malware, these systems have been limited by incomplete view of inspection on a single layer. What's even worse, various new techniques including packing, anti-emulator and cross-layer malicious payloads, employed by the latest malware samples further make these systems ineffective. In this thesis, we propose Malton, a novel on-device non-invasive analysis platform for the new Android runtime, i.e., the ART runtime. Malton runs on real mobile devices and provides a comprehensive view of malware's behaviours by conducting multi-layer monitoring and information flow tracking, as well as efficient path exploration. We have carefully evaluated Malton using real-world malware samples. The experimental results showed that Malton is more effective than the existing tools, with the capability to analyze sophisticated malware samples and provide a comprehensive view of malicious behaviours of these samples.

In summary, the following contributions are what we make in Malton.

- We proposed a novel Android malware analysis system, with the capability to provide a comprehensive view of malicious behaviours. It employs two capabilities, i.e., multi-layer monitoring and information flow tracking, and efficient path exploration, which are not available on the existing systems.
- We implemented the system based on Valgrind and solved several technical challenges. To engage the whole community, we plan to release the Malton system to the community.
- We carefully evaluated Malton with real-world malware samples. The results demonstrated the effectiveness of Malton in analyzing sophisticated malware.

The rest of this chapter is organized as follows. Chapter 3.2 introduces background knowledge and describes a motivating example. Chapter 3.3 details the system design and implementation. Chapter 3.4 reports the evaluation results.

3.2 Background

3.2.1 The ART Runtime

ART is the new runtime introduced in Android version 4.4, and becomes the default runtime from version 5.0. When an app is being installed, its Dalvik bytecode in the Dex file is compiled to native code¹ by the `dex2oat` tool, and a new file in the OAT format is generated including both the Dalvik bytecode and native code. The OAT format is a special ELF format with some extensions.

The OAT file has an `oatdata` section, which contains the information of each class

¹ Native code denotes the native instructions that could directly run with a particular processor.

that has been compiled into native code. The native code resides in a special section with the offset indicated by the `oatexec` symbol. Hence, we can find the information of a Java class in the `oatdata` section and its compiled native code through the `oatexec` symbol.

When an app is launched, the ART runtime parses the OAT file and loads the file into memory. For each Java class object, the ART runtime has a corresponding instance of the C++ class `Object` to represent it. The first member of this instance points to an instance of the C++ class `Class`, which contains the detailed information of the Java class, including the fields, methods, etc.. Each Java method is represented by an instance of the C++ class `ArtMethod`, which contains the method's address, access permissions, the class to which this method belongs, etc.. The C++ class `ArtField` is used to represent a class field, including the class to which this field belongs, the index of this field in its class, access rights, etc.. We can leverage the C++ `Object`, `Class`, `ArtMethod` and `ArtField` to find the detailed information of the Java class, methods and fields of the Java class.

The Android framework is compiled into an OAT file named “*system@framework@boot.oat*”. This file is loaded to the fixed memory range for all apps running on the device without ASLR enabled [160].

3.2.2 A Motivating Example

We use the example in Listing 3.1 to illustrate the usage of Malton. In this example, the method `onReceiver()` is a SMS listener and it is invoked when an SMS arrives. In this method, the telephone number of the sender is first acquired (Line 39) for checking whether the SMS is sent from the controller (Tel: 6223**60). Only the SMS from the controller will be processed by the method `procCMD()` (Line 42). There are 5 types of

Listing 3.1: A motivating example

```
1 public static native void readContact();
2 public static native void parseMSG(String msg);
3 private void readIMSI(){
4     TelephonyManager telephonyManager =
5         (TelephonyManager) getSystemService(
6             Context.TELEPHONY_SERVICE);
7     String imsi = telephonyManager.getSubscriberId();
8     // Send back data through SMTP protocol
9     smtpReply(imsi);
10 }
11 private void procCMD(int cmd, String msg){
12     if(cmd == 1) {
13         readSMS(); // Read SMS content
14     } else if(cmd == 2) {
15         readContact(); // Read Contact content
16     } else if(cmd == 3) {
17         readIMSI(); // Read device IMSI information
18     } else if(cmd == 4) {
19         rebootDevice(); // Reboot the device
20     } else if(cmd == 5) {
21         parseMSG(msg); // Parse msg in native code
22     } else { // The command is unrecognized.
23         reply("Unknown command!");
24     }
25 }
26 public boolean equals(String s1, String s2) {
27     if(s1.count != s2.count)
28         return false;
29     if(s1.hashCode() != s2.hashCode())
30         return false;
31     for(int i = 0; i < count; ++i)
32         if (s1.charAt(i) != s2.charAt(i))
33             return false;
34     return true;
35 }
36 public void onReceiver(Context context, Intent intent){
37     String body = smsMessage.getMessageBody();
38     // Get the telephone of the sender
39     String sender = smsMessage.getOriginatingAddress();
40     // Check if the SMS is sent form the controller
41     if(equals(sender, "6223**60")) {
42         procCMD(Integer.parseInt(body), body);
43     }
44     ...
45 }
```

commands, each leads to a special malicious behaviour (i.e., Line 13, 15, 17, 19 and 21).

Reading contact and parsing SMS are implemented in the JNI methods *readContact()* (Line 1) and *parseMSG()* (Line 2), respectively.

Existing malware analysis tools could not construct a complete view of the malicious behaviours. For example, when *cmd* equals 3 (Line 16), IMSI is obtained by invoking the framework API *getSubscriberId()* (Line 7), and then leaked through SMTP protocol (Line 9). Although existing tools (e.g., CopperDroid[163]) can find that the malware reads

IMSI and leaks the information by system call *sys_sendto()*, they cannot locate the method used to get IMSI and how the IMSI is leaked in detail, because *sys_sendto()* can be called by many functions (e.g., JavaMail APIs, Java Socket methods and C/C++ Socket methods) from both the framework layer and the native layer. Malton can solve this problem because it performs multi-layer monitoring.

When *cmd* equals 5, the content of SMS, which is obtained from the framework layer (Line 37), will be parsed by the JNI method *parseMSG()* (Line 2) in the native code. Although taint analysis could identify this information flow, existing static instrumentation based tools (e.g., TaintART[161] and ARTist[53]) cannot track the information flow in the native code. Malton can tackle this issue since it offers cross-layer taint analysis.

Moreover, as shown in the method *procCMD()* (Line 11), the malware performs different activities according to the parameter *cmd*. Due to the low code coverage of dynamic analysis, how to efficiently explore all the malicious behaviours with the corresponding inputs is challenging. Malton approaches this challenge by conducting concolic execution with in-memory optimization and directed sexecution. Furthermore, we propose a new offloading mechanism to avoid overloading the mobile devices with limited computational resources. Since some constraints may not be solved (e.g., the hash functions at Line 29), we develop a directed execution engine to cover specified branches forcibly.

3.3 Malton

In this section, we firstly illustrate the design of our approach, and then we detail the implementation of Malton.

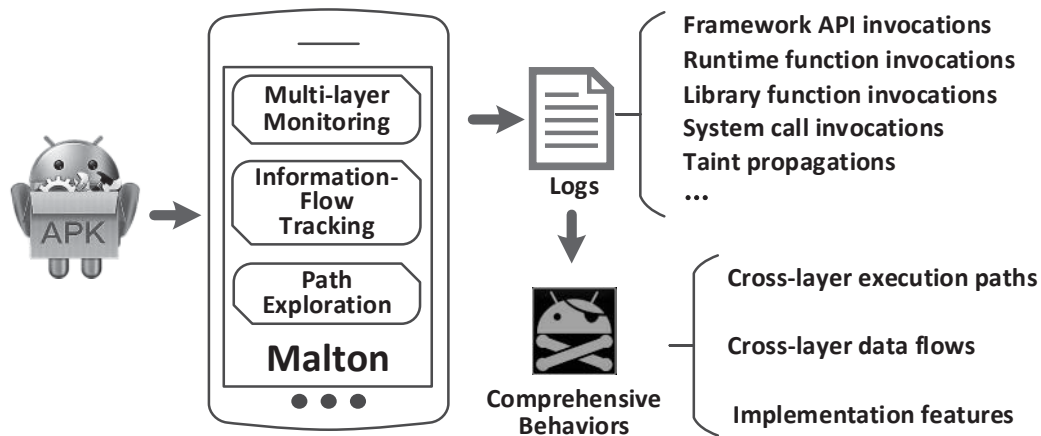


Figure 3.1: The scenario of Malton.

3.3.1 Overview

Malton helps security analysts obtain a complete view of malware samples under examination. To achieve this goal, Malton supports three major functionalities. First, due to the multi-layer nature of Android system, Malton captures malware behaviours at different layers. For instance, malware may conduct malicious activities by invoking native code from Java methods, and such behaviours involve method invocations and data transmission at multiple layers. The challenging issue is how to effectively bridge the semantic gap when monitoring the ARM instructions.

Second, malware could leak private information by passing the data across multiple layers back and forth. Note that many frameworks APIs are JNI methods (e.g., *String.concat()*, *String.toCharArray()*, etc.), whose real implementations are in native code. Malton can detect such privacy leakage because it supports cross-layer information flow tracking (Chapter 3.3.5).

Third, since malware may conduct diverse malicious activities according to different commands and contexts, Malton can trigger these activities by exploring the paths automatically (Chapter 3.3.6). It is not trivial to achieve this goal because dynamic analysis

systems usually have limited code coverage.

Figure 3.1 illustrates the usage scenario of Malton. Malton runs in real Android devices and conducts multi-layer monitoring, information flow tracking, and path exploring. After running a malware sample, Malton generates logs containing the information of method invocations, taint propagations at different layers and the result of concolic executions. Based on the logs, we reconstruct the execution paths and the information flows for characterizing malware behaviours.

Though Malton performs the analysis in multiple layers as shown in Figure 3.2, the implementation of Malton in each layer is not independent. Conversely, different layers share the information with each other. For example, the taint propagation module in the instruction layer needs the information about the Java methods that are parsed and processed in the framework layer.

Malton is built upon Valgrind [126] (V3.11.0) with around 25k lines of C/C++ codes calculated by CLOC [5]. Next, we will detail the implementation at each layer.

3.3.2 Android Framework Layer

To monitor the invocations of privacy-concerned Java methods of the Android framework and the app itself, Malton instruments the native code of the framework and the app. Since the Dalvik code has been compiled into native instructions, we leverage Valgrind for the instrumentation. The challenge here is how to recover and understand the semantic information of Java methods from the ARM instructions, including the method name, parameters, call stacks, etc. For instance, if a malware sample uses the Android framework API to retrieve user contacts, Malton should capture this behaviour from the ARM instructions and recover the context of the API invocation. To address this challenge,

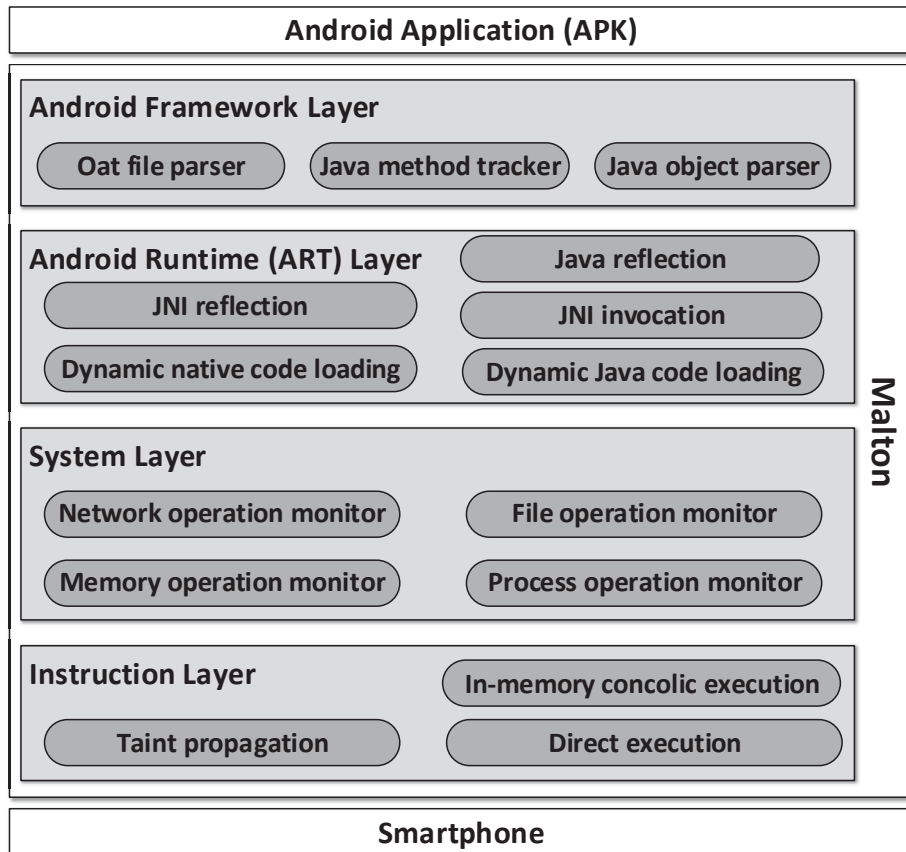


Figure 3.2: The overview of the system implementation.

we propose an efficient way to bridge the semantic gap between the low level native instructions and upper layer Java methods.

Java Method Tracker To track the Java method invocations, we need to identify the entry point and exit points of each Java method from the ARM instructions dynamically. Note that the ARM instructions resulted from the Dalvik bytecode are further translated into multiple IR blocks by Malton. An IR block is a collection of IR statements with one entry point and multiple exit points. One exit point of an IR block could be either the conditional exit statement (i.e., `Ist_Exit`) or the next statement (i.e., `Ist_Next`). We leverage the APIs from Valgrind to add instrumentation at the beginning, before any IR instruction, after any IR instruction, or at the end of the selected IR block. The instrumentation

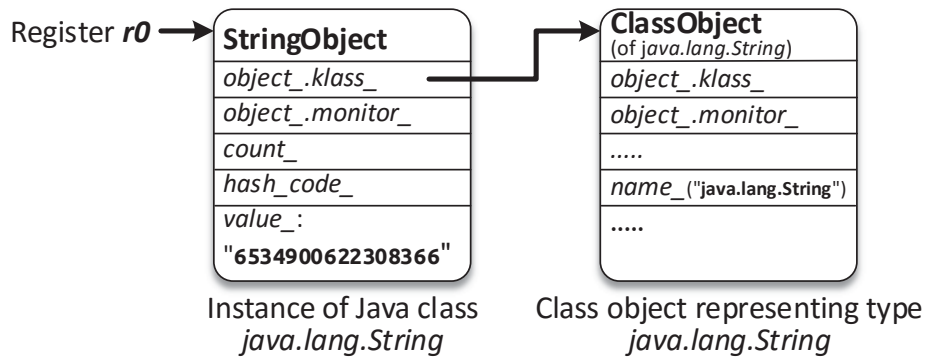


Figure 3.3: The example to parse the result of `TelephonyManager.getDeviceId()`

statements will invoke our helper functions.

To obtain the entry point of a Java method, we use the method information in the OAT file. Specifically, the OAT file contains the information of each compiled Java method (`ArtMethod`), including the method name, offset of the ARM instructions, access flags, etc.. Malton parses the OAT files of both the Android framework and the app itself to retrieve such information, and keeps it in a hash table. When the native code is translated into the IR blocks, Malton looks up the beginning address of each IR block in the hash table to decide whether it is the entry of a Java method. If so, Malton inserts the helper function (i.e., `callTrack()`) at the beginning of the block to record the method invocation and parse arguments when it is executed.

To identify the exit point of a Java method, Malton leverages the method calling convention of the ARM architecture². Specifically, the return address of a method is stored in the link register (i.e., the `lr` register) when the method is invoked. Hence, in `callTrack()`, Malton pushes the value of `lr` into the method call stack since `lr` could be changed during the execution of the method. Malton also inserts the helper function (i.e., `retTrack()`) before each exit point (i.e., `Ist_Exit` and `Ist_Next`) of the IR block. In `retTrack()`, Malton compares the jump target of the IR block with the method's return address stored at the

² Comments in file `/art/compiler/dex/quick/arm/arm.lir.h`

top of the method call stack. If they are equal, an exit point of the method is found, and this return address is popped from the method call stack.

Malton parses the arguments and the return value of the method after the entry point and the exits point of the method are identified, respectively. According to the method calling convention, the register *r0* points to the `ArtMethod` object of current method, and registers *r1* – *r3* contain the first three arguments. Other arguments beyond the first three words are pushed into the stack by the caller. For example, when the framework method `sendMessageAtTime(Message msg, long uptimeMillis)` of class `android/os/Handler` is invoked, *r0* points to the `ArtMethod` instance of the method `sendMessageAtTime()`, *r1* stores the `this` object and *r2* represents the argument `msg`. For the argument `uptimeMillis`, the high 32 bits are stored in the register *r3* and the low 32 bits are pushed into the stack. When the method returns, the return value is stored in the register *r0* if the return value is 32 bits, and in registers *r0* and *r1* if the return value is 64 bits.

Java Object Parser After getting the method arguments and the return value, we need to further parse the value if it is not the primitive data. There are two major data types [9] in Java, including primitive data types and reference/object data types (objects). For the primitive types, which include byte, char, short, int, long, float, double and boolean, we can directly get the value from registers and the stack. For the object, the value that we obtain from the register or the stack is a pointer that points to a data structure containing the detailed information of this object. Following this pointer, we get the class information of this object, and then parse the memory of this object to determine the concrete value.

Figure 3.3 illustrates the process of parsing the Java object of the result of `TelephonyManager.getDeviceId()`. According to its method shortly, we know that the return value of this API is a Java object represented by an `Object` instance, of which the memory address is stored in the register *r0*. Then, we can decide that the concrete type of this

Table 3.1: Runtime behaviours related functions.

Behaviours	Functions
Native code loading	<i>JavaVMExt::LoadNativeLibrary()</i>
Java code loading	<i>DexFile::DexFile()</i> <i>DexFile::OpenMemory()</i> <i>ClassLinker::DefineClass()</i>
JNI invocation	<i>artFindNativeMethod()</i> <i>ArtMehod::invoke()</i>
JNI reflection	<i>InvokeWithVarArgs()</i> <i>InvokeWithJValues()</i> <i>InvokeVirtualOrInterfaceWithJValues()</i> <i>InvokeVirtualOrInterfaceWithVarArgs()</i>
Java reflection	<i>InvokeMethod()</i>

object is *java.lang.String*. By parsing the results according to the memory layout of *String* object, which is represented by the *StringObject* data structure, we can obtain the concrete string “6534900622308366”. Currently, Malton parses the Java objects related to *String* and *Array*. To handle new objects, users just need to implement the corresponding parsers for Malton.

3.3.3 Android Runtime Layer

To capture stealthy behaviours that cannot be monitored by the Java method tracker in the Android framework layer, Malton further instruments the ART runtime (i.e., *libart.so*). For example, the packed malware may use the internal functions of the ART runtime to load and execute the decrypted bytecode directly from the memory [193, 182]. Malicious payloads could also be implemented in native code, and then invoke the privacy-sensitive Java methods from native code through the JNI reflection mechanism. While the invoked Java method could be tracked by the Java method tracker in the Android framework layer, Malton tracks the JNI reflection to provide a comprehensive view of malicious behaviours, such as, the context when privacy-concerned Java methods are invoked from the native

code. This is one advantage of Malton over other tools.

Table 3.1 enumerates the runtime behaviours and the corresponding functions in the ART runtime that Malton instruments. Native code loading means that malicious code could be implemented in native code and loaded into memory, where Java code loading refers to loading the Dalvik bytecode. Note that Android packers usually exploit these APIs to directly load the decrypted bytecode from memory. JNI invocation refers to all the function calls from Java methods to native methods. This includes the JNI calls in the app and the Android framework. JNI reflection, on the other hand, refers to calling Java methods from native code. For instance, malicious payloads implemented in native code could invoke framework APIs using JNI reflection. Java reflection is commonly used by malware to modify the runtime behaviour for evading the static analysis [141]. For example, framework APIs could be invoked by decrypting the method names and class names at runtime using Java reflection.

3.3.4 System Layer

Malton tracks system calls and system library functions at the system layer. To track system calls, Malton registers callback handlers before and after the system call invocation through Malton APIs. For system library functions, Malton wraps them using the function wrapper mechanism of Valgrind. In the current prototype, Malton focuses on four types of behaviours at the system level.

- Network operations. Since malware usually receives the control commands and sends private data through network, Malton inspects these behaviours by wrapping network related system calls, such as, *sys_connect()*, *sys_sendto()*, *recvfrom()*, etc..
- File operations. As malware often accesses sensitive information in files and/or

Table 3.2: The taint propagation related IR Statements.

IR Statement	Representation
Ist_WrTmp	Assign a value (i.e., IR Expression) to a temporary.
Ist_LoadG	Load a value to a temporary with guard.
Ist_CAS	Do an atomic compare-and-swap operation.
Ist_LLSC	Do an either load-linked or store-conditional operation.
Ist_Put	Write a value to a guest register.
Ist_PutI	Write a value to a guest register at a non-fixed offset in the guest state.
Ist_Store	Write a value to memory.
Ist_StoreG	Write a value to memory with guard.
Ist_Dirty	Call a C function.

dynamically loads malicious payloads from the file system, Malton records file operations to identify such behaviours.

- **Memory operations.** Since packed malware usually dynamically modifies its own codes through memory operations, like *sys_mmap()*, *sys_protect()*, etc., Malton monitors such memory operations.
- **Process operations.** As malware often needs to fork new process, or exits when the emulator or the debug environment is detected, Malton captures such behaviours by monitoring system calls relevant to the process operations, including *sys_execve()*, *sys_exit()*, etc..

Moreover, Malton may need to modify the arguments and/or the return values of system calls to explore code paths. For example, the C&C server may have been shut down when malware samples are being analyzed. In this case, Malton replaces the results of the system call *sys_connect()* to success, or replaces the address of C&C with a bogus one controlled by the analyst to trigger malicious payloads. We will discuss the techniques used to explore code paths in Chapter 3.3.6.

Table 3.3: Taint propagation related IR expressions.

IR Expression	Representation
Iex_Const	A constant-valued expression.
Iex_RdTmp	The value held by a VEX temporary.
Iex_ITE	A ternary if-then-else operation.
Iex_Get	Get the value held by a guest register at a fixed offset.
Iex_GetI	Get the value held by a guest register at a non-fixed offset.
Iex_Unop	A unary operation.
Iex_Binop	A binary operation.
Iex_Triop	A ternary operation.
Iex_Qop	A quaternary operation.
Iex_Load	Load the value stored in memory.
Iex_CCall	A call to a pure (no side-effects) helper C function.

3.3.5 Instruction Layer: *Taint Propagation*

At the instruction layer, Malton performs two major tasks, namely, taint propagation and path exploration. Note that accomplishing these tasks needs the semantic information in the upper layers, such as the method invocations for identifying the information flow, etc..

To propagate taint tags across different layers, Malton works at the instruction layer because the codes of all upper layers become ARM instructions during the execution. Since these ARM instructions will be translated into IR statements [126], Malton performs taint propagation on IR statements with byte precision by inserting helper functions before the selected IR statements.

For Malton, there are 9 types IR statements related to the taint propagation, which are listed in Table 3.2. For the `Ist_WrTmp` statement, since the source value may be the result of an IR expression, we also need to parse the logic of the IR expression for taint propagation. The IR expressions that can affect the taint propagation are summarized in Table 3.3. During the execution of the target app, Malton parses the IR statements and

expressions in the helper functions, and propagates the taint tags according to the logic of the IR statements and expressions.

Malton supports taint sources/sinks in different layers (i.e., the framework layer and the system layer). For example, Malton can take the arguments and results of both Java methods and C/C++ methods as the taint sources, and check the taint tags of the arguments and the results of sink methods. By default, at the framework layer, 11 types of information are specified as taint sources, including device information (i.e., IMSI, IMEI, SN and phone number), location information (i.e., GPS location, network location and last seen location) and personal information (i.e., SMS, MMS, contacts and call logs). Malton also checks the taint tags of the arguments and results when each framework method is invoked. In the system layer, Malton takes system calls `sys_write()` and `sys_sendto()` as taint sinks by default, because the sensitive information is usually stored to files or leaked out of the device through these system calls. As malware can receive commands from network, Malton takes system call `sys_recvfrom()` as the taint source by default. Note that Malton can be easily extended to support other methods as taint sources and sinks in both the framework layer and the system layer.

3.3.6 Instruction Layer: *Path Exploration*

Advanced malware samples usually execute malicious payloads according to the commands received from the C&C server or the special context (e.g., date, locations, etc.). To trigger as many malicious behaviours as possible for analysis, Malton employs the efficient path exploration technique, which consists of taint analysis, in-memory concolic execution with an offloading mechanism, and directed execution engine. Specifically, taint analysis helps the analyst identify the code paths depending on the inputs, such as network commands, and the concolic execution module can generate the required inputs to explore

the interested code paths. When the inputs cannot be generated, we rely on the directed execution engine to forcibly execute certain code paths. Since concolic execution [66] is a well-known technique in the community, we will not introduce it in the following. Instead, we detail the offloading mechanism and the in-memory optimization used in the concolic execution module, and explain how the directed execution engine works.

Concolic Execution: Offloading Mechanism It is non-trivial to apply concolic execution in analyzing Android malware on real devices, because concolic execution requires considerable computational resources, resulting in unacceptable overhead on the mobile devices. To alleviate this limitation, Malton utilizes an offloading mechanism that moves the task of solving constraints to the resourceful desktop computers, and then sends back the satisfying results to the mobile devices as inputs. Our approach is motivated by the fact that the time consumption for solving constraints occupies the overall runtime of concolic execution. For example, the percentage of time used to solve constraints is nearly 41% of the KLEE system, even after optimizations [61].

More precisely, when the malware sample is running in our system, Malton redirects all the constraints to the logcat messages [12], which could be retrieved by the desktop computer using the ADB (Android Debug Bridge) tool. Then, the constraint solver, which is implemented based on Z3 [76], generates the satisfying inputs and feeds the inputs back to Malton through a file. Since we may have multiple code paths that need to be explored, this process could be repeated multiple times until the constraint solver pushes an empty input file to the device for notifying Malton to finish path exploring.

Concolic Execution: In-memory Optimization To speed up the analysis, especially when there are multiple execution paths, each of which depends on the special input, we propose in-memory optimization to restrict concolic execution within the interested code region specified by the analyst without repeatably running from the beginning of

the program. By default, the analyst is required to specify the arguments or variables as the input of the concolic execution, which will be represented as symbolic values during concolic execution. For example, the analyst can select the SMS content acquired from the method *getMessageBody()* (Line 37 in Listing 3.1) as the input. Moreover, the analyst can select the IR statement that lets the input have concrete values as the entry point of the code region, and choose the exit statement (i.e., `Ist_Exit`) or the next statement (i.e., `Ist_Next`) of the subroutine as the exit point of the code region.

Malton runs the malware sample until the exit point of the interested code region for collecting constraints and generating new inputs for different code paths through an SMT solver. Then, it forces the execution to return to the entry point of the code region through modifying the program counter and feeds the inputs by writing the new inputs directly into the corresponding locations (i.e., memories or registers). Moreover, Malton needs to recover the execution context and the memory state at the entry point of the code region.

To recover the execution context, Malton conducts instrumentation at the beginning of the code region, and inserts a helper function to save the execution context (i.e., register states at the first iteration). After that, the saved register states will be recovered in the later iterations. As Valgrind uses the structure `VexGuestArchState` to represent the register states, we save and recover the register states by reading and writing the `VexGuestArchState` data in the memory.

To recover the memory states, Malton replaces the system's memory allocation/free functions with our customized implementations to monitor all the memory allocation/free operations. Malton can also free the allocated memory or re-allocate the freed memory. Besides, Malton inserts a helper function before each memory store (i.e., `Ist_Store` and `Ist_StoreG`) statement to track the memory modifications, so that all the modified memory could be restored.

Alternatively, the analyst can choose the target code region according to the method call graph, or first use static analysis tool to identify code paths and then select a portion of the path as the interested code region.

Directed Execution The concolic execution may not be able to explore all the code paths of the interested code region, because the constraint solver may not find satisfying inputs for complex constraints, such as float-point operations and encryption routines. For the conditional branches with unresolved constraints, Malton has the capability to directly execute certain code paths.

The directed execution engine of Malton is implemented through two techniques: a) modifying the arguments and the results of methods, including library functions, system calls and Java methods; b) setting the guard value of the conditional exit statement (i.e., `Ist_Exit`). The guard value is the expression used in the `Ist_Exit` statement to determine whether the branch should be taken.

It's straightforward to modify arguments and the return values of library functions and system calls by leveraging Valgrind APIs. However, it's challenging to deal with the Java methods because there is no interface in Valgrind to wrap Java methods. Fortunately, we have obtained the entry point and exit points of the compiled Java method in the framework layer (Chapter 3.3.2). Hence, we could wrap the Java method by adding instrumentation at its entry point and exit points. For example, to change the source telephone number of a received SMS to explore certain code path (Line 41 in Listing 3.1), Malton can wrap the framework API `SmsMessage.getOriginatingAddress()` and modify its return value to a desired number at the exit points.

To set the guard value of the `Ist_Exit` statement, we insert a helper function before each `Ist_Exit` statement and specify the guard value to the result of the helper function.

In an IR block, the program can only conventionally jump out of the IR block at the location of the `Ist_Exit` statement (e.g., an if-branch in the program). The `Ist_Exit` statement is defined with the format “`if(t) goto <dst>`” in Valgrind, where *t* and *dst* represent the guard value and destination address, respectively. By returning “1” or “0” in the helper function, we let *t* satisfy or dissatisfy the condition for exploring different code paths.

3.4 Evaluation

We evaluate Malton on real-world Android malware samples to answer the following questions.

Q3.1: Whether Malton could capture more sensitive operations than the existing tools, due to the multi-layer monitoring capability?

Q3.2: Whether Malton could analyze sophisticated malware samples, including the packed ones, to provide a comprehensive view of malicious behaviours?

Q3.3: Whether the path exploration mechanism is effective and efficient?

3.4.1 Sensitive Behavior Monitoring

To answer **Q3.1**, we compare Malton’s capability of capturing sensitive behaviours with CopperDroid [163] and DroidBox [79]. These two systems are implemented by instrumenting Android emulator and modifying the Android framework, respectively. Since CopperDroid’s website³ has just queued all our uploaded malware samples, we cannot obtain the corresponding analysis results. Therefore, we downloaded the analysis

³ <http://copperdroid.isg.rhul.ac.uk/copperdroid/reports.php>

reports of 1,362 malware samples that have been analyzed by CopperDroid. According to their md5s, we have collected 512 samples, and run them using Malton and DroidBox, respectively. The comparison results are listed in Table 3.4. The first column shows the type of sensitive behaviours, and the following columns list the numbers and percentages of malware samples that have been detected by each system due to the corresponding sensitive behaviours. We can see that for all the sensitive behaviours Malton detected more samples than the other two systems.

We further manually analyze the malware samples to understand why Malton detects more sensitive behaviours in those samples than the other two systems. First, Malton monitors malware's behaviours in multiple layers, and thus it captures more behaviours than the systems focusing on one layer. For instance, the malware sample⁴ retrieves the serial number and operator information of the SIM card through the framework APIs *TelephonyManager.getSimSerialNumber()* and *TelephonyManager.getSimOperator()*, respectively. However, CopperDroid does not support reconstructing such behaviours from system calls and DroidBox does not monitor these framework APIs. Second, Malton runs on real devices, and hence it could circumvent many anti-emulator techniques. For instance, the malware sample⁵ detects the existence of emulator based on the value of *android_id* and *Build.DEVICE*. If the obtained value indicates that it is running in an emulator, the malicious behaviours will not be triggered.

Note that these samples were analyzed by CopperDroid before 2015 and it is likely that their C&C servers were active at that time. However, not all C&C servers were still active when Malton inspects the same samples. Hence, in the worst case, Malton's results may be penalized since the malware cannot receive commands.

⁴ md5: 021cf5824c4a25ca7030c6e75eb6f9c8

⁵ md5: a0000a85a2e8e458660c094ebedc0c6e

Table 3.4: Comparison on the capability of capturing the sensitive behaviours of malware samples.

Behavior	CopperDroid	DroidBox	Malton
Personal Info	435 (85.0%)	135 (26.4%)	511 (99.8%)
Network access	351 (68.5%)	211 (41.2%)	445 (86.9%)
File access	438 (85.5%)	509 (99.4%)	512 (100%)
Phone call	52 (10.1%)	1 (0.2%)	59 (11.5%)
Send SMS	26 (5.1%)	15 (2.9%)	28 (5.5%)
Java code loading	Na	509 (99.4%)	512 (100%)
Anti-debugging	4 (0.8%)	Na	4 (0.8%)
Native code loading	Na	Na	160 (31.2%)

Summary Compared with existing tools running in the emulator and monitoring malware behaviours in a single layer, Malton can capture more sensitive behaviours thanks to its on-device and cross-layer inspection.

3.4.2 Malware Analysis

To answer **Q3.2**, we evaluate Malton with sophisticated malware samples by constructing the complete flow of information leakage across different layers, detecting stealthy behaviours with Java/JNI reflection, dissecting the behaviours of packed Android malware, and identifying the malicious behaviours of hidden code.

Identifying Information Leakage Flow This experiment uses the sample in the XXShenqi [8] malware family, which is an SMS phishing malware with package name *com.example.xxshenqi*. When the malware is launched, it reads the contact information and creates a phishing SMS message that will be sent to all the contacts collected. In this inspection, we focused on the behaviour of creating and sending the phishing SMS message to the retrieved contacts by letting the contacts be the taint source and the methods for sending SMS messages be the taint sink. The detailed flow is illustrated in Figure 3.4.

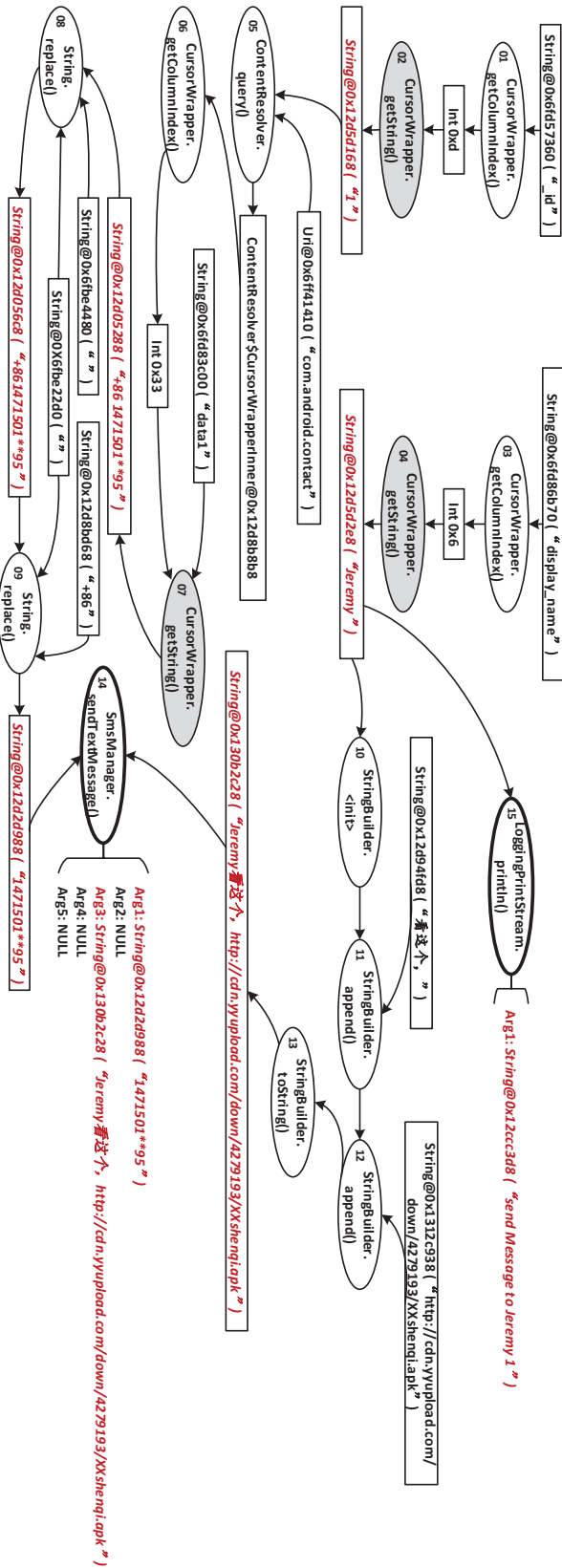


Figure 3.4: The detailed flow of contact related information tracking of the XXshengqi malware (Grey diagrams represent taint sources, diagrams with bold lines represent taint sinks, rectangles represent data, ellipses represent behaviours (methods) and red italics strings represent tainted information).

To retrieve the information of each contact, the malware first obtains the column index and the value of the field `_id` in step 1 and step 2 in Figure 3.4⁶, respectively. Then, a new instance of the class `CursorWrapper` is created based on `_id` and `uri` (`com.android.contact`), and this contact's phone number is acquired through this instance. After that, blank characters and the national number (“+86”) are removed from the retrieved phone number in steps 8 and 9. In the method `String.replace()`⁷, `StringFactory.newStringFromString()` and `String.setCharAt()` are invoked to create a new string according to the current string and set the specified character(s) of the new string, respectively. These two methods are JNI functions and implemented in the system layer. For `String.setCharAt()`, Malton can further determine the tainted portion of the string at the byte granularity. By contrast, TaintDroid does not support this functionality because for JNI methods it lets the taint tag of the whole return value be the union of the function arguments' taint tags. After that, a phishing SMS message is constructed according to the `display_name` of a retrieved contact and the phishing URL through steps 10-13. Finally, the phishing SMS is sent to the contact in step 14 and a message “send Message to Jeremy 1” is printed in step 15.

Summary By conducting the cross-layer taint propagation, Malton can help the analyst construct the complete flow of information leakage.

Detecting Stealthy Behavior Identification Some malware adopts Java/JNI reflection to hide their malicious behaviours. We use the sample in the `photo3`⁸ malware family to evaluate Malton's capability of detecting such stealthy behaviours. Figure 3.5 demonstrates the identified stealthy behaviours, which are completed in two different threads. The number in the ellipse and rectangle is the step index, and we use different

⁶ The number in each ellipse denotes the step index.

⁷ in `/libcore/libart/src/main/java/java/lang/String.java`

⁸ `md5:8bd9f5970afec4b8e8a978f70d5e87ab`

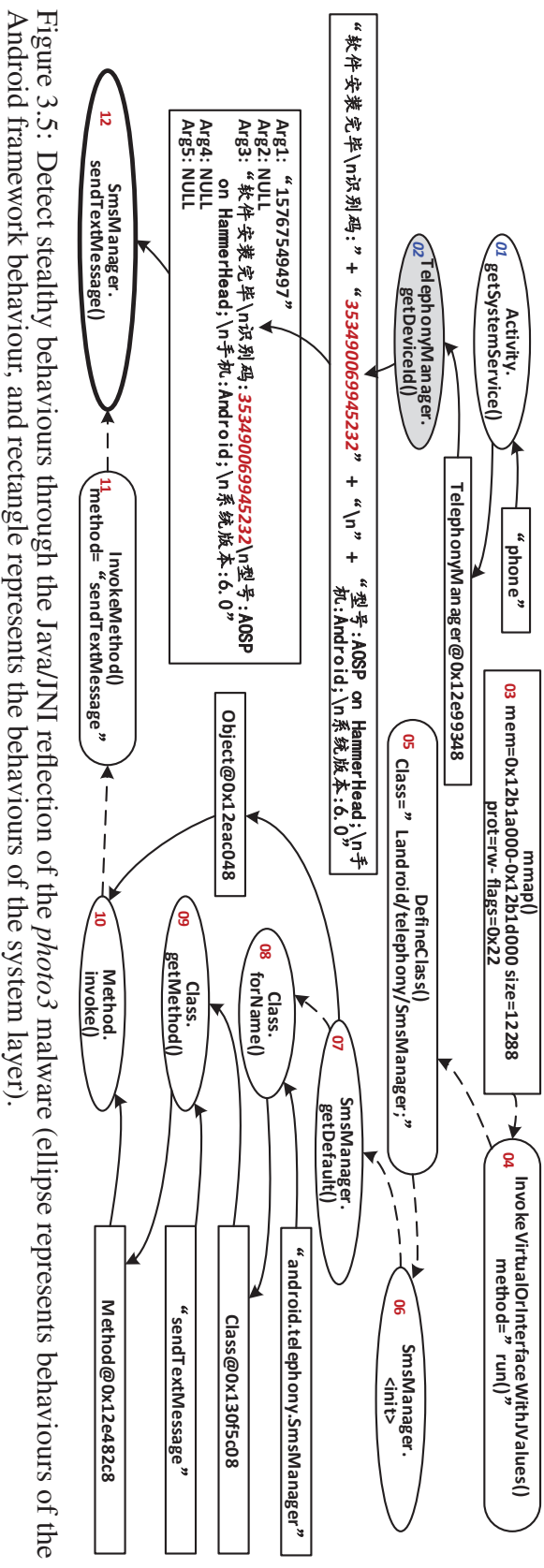


Figure 3.5: Detect stealthy behaviours through the Java/INI reflection of the *photo3* malware (ellipse represents behaviours of the Android framework behaviour, and rectangle represents the behaviours of the system layer).

colours (i.e., blue and red) for the numbers to distinguish two threads. The execution paths are denoted by both the solid lines and dashed lines, and the solid lines further indicate how the information is leaked. We describe the identified malicious behaviours as follows.

- The device ID is returned by the method *TelephonyManager.getId()* in step 1 and step 2.
- A new thread is created to send the collected information to the malware author. In step 3, a memory area is allocated by the system call *sys_mmap()*, and the thread method *run()* is invoked by the runtime through the JNI reflection function *InvokeVirtualOrInterfaceWithJValues()* in step 4. Next, the class *android/telephony/SmsManager* is defined and initialized in step 5 and step 6. In step 7, the *SmsManager* object is obtained through the static method *SmsManager.getDefault()*.
- The malware sends SMS messages through Java reflection. Specifically, in step 8, the malware obtains the object of the *android.telephony.SmsManager* class through the Java reflection method *Class.forName()*. Then, it retrieves the method object of *sendTextMessage()* using the Java reflection method *Class.getMethod()* in step 9. Finally, it calls the Java method *sendTextMessage()* in step 10. This invocation goes to the method *InvokeMethod()* in the ART runtime layer in step 11.

Summary Malton can identify malware’s stealthy behaviours through Java/JNI reflection in different layers.

Dissecting Packed Android Malware’s Behaviors Since Malton stores the collected information into log files, we can dissect the behaviours of packed Android malware by analyzing the log files. As an example, Figure 3.6 shows partial log file of analyzing the packed malware sample⁹, and Figure 3.7 illustrates the identified malicious behaviours of this sample. Such behaviours can be divided into two parts. One is related to the

⁹ md5: 03b2deeb3a30285b1cf5253d883e5967

```

Behaviors of "com.netease.nis.wrapper.MyApplication"
01 Instrumentation.newApplication()
02 ClassLoader.loadClass("com.netease.nis.wrapper.MyApplication")
03 Application.init()
04 Application.attach() // Internal framework API
05 ContextWrapper.attachBaseContext() // Set the base context for this ContextWrapper.
06 ...; // Malicious behaviors 1
07 System.loadLibrary("nsec") // Load native library libnsec.so
08 FindClass("com/netease/nis/wrapper/MyJni") // Find and define Class = "com/netease/nis/wrapper/MyJni"
09 LoadNativeLibrary("/data/app/com.vnuhqwdqddq.trarenren5-1/lib/arm/libnsec.so") // Load library libnsec.so
10 MyJni.load() // Invoke the JNI method MyJni.load()
11 InvokeVirtualOrInterfaceWithVarArgs() // JNI reflection invocation. args: Method=Context.getPackageName()
12 Context.getPackageName() // res: "com.vnuhqwdqddq.trarenren5"
13 sys_open("/data/data/com.vnuhqwdqddq.trarenren5/cache/classes.dex") // res: fd = 24
14 sys_write(fd = 24); sys_close(fd = 24) // Write protected dex content to classes.dex
15 /* Open and initialize DexFile arg : location="/data/user/0/com.vnuhqwdqddq.trarenren5/cache/classes.dex" */
16 OpenMemory() // res: DexFileObj@0x06d541c8 The DexFile object is used to represented the dex file in Android runtime
Behaviors of "v.v.v.MainActivity"
17 Instrumentation.callApplicationOnCreate() // arg: Application@0x12e05498
18 Application.onCreate() // Called when the application is starting, before the activity is created
19 IntentFilter.<init>("com.zjdroid.invoke") // Create an IntentFilter@0x12e4d848,
20 /* Register an Intent receiver dynamically */
21 ContextWrapper.registerReceiver() // arg: IntentFilter@0x12e4d848
22 Instrumentation.newActivity() // Initialize the new activity arg: Activity="v.v.v.MainActivity", res: Activity@0x12c79f08
23 ClassLoader.loadClass("v.v.v.MainActivity") // Load Class="v.v.v.MainActivity", res: Class@0x13110808
24 DefineClass() // args: DexFileObj=0x06d541c8 Class="Lv/v/v/MainActivity"
25 Class.newInstance()
26 Activity.init()
27 Instrumentation.callActivityOnCreate() // Create and display an activity
28 Activity.performCreate() // Create activity "v.v.v.MainActivity"
29 ...; // Malicious behaviors 2
30 Activity.finish() // Close the activity for hiding

```

Figure 3.6: The major information collected by Malton at function level. The names of Android runtime functions and system calls are in black italics. We omit the information of method arguments due to the limited space.

original packed malware (Lines 1-21), and the other is relevant to the hidden payloads of the malware (Lines 22-30).

Once the malware is started, the class *com.netease.nis.wrapper.MyApplication* is loaded for preparing for the real payload (Line 2). Then, the Android framework API *Application.attach()* is invoked (Line 4) to set the property of the app context. After that, the malware calls the Java method *System.loadLibrary()* to load its native component *libnsec.so* at Line 7. Malton empowers us to observe that the ART runtime invokes the function *FindClass()* (Line 8) and the function *LoadNativeLibrary()* (Line 9) to locate the class *com.netease.nis.wrapper.MyJni* and load the library *libnsec.so*, respectively.

After initialization, the malware calls the JNI method *MyJni.load()* to release and load the hidden Dalvik bytecode into memory. More precisely, the package name is first obtained through JNI reflection (Line 11 and 12). Then, the hidden bytecode is written

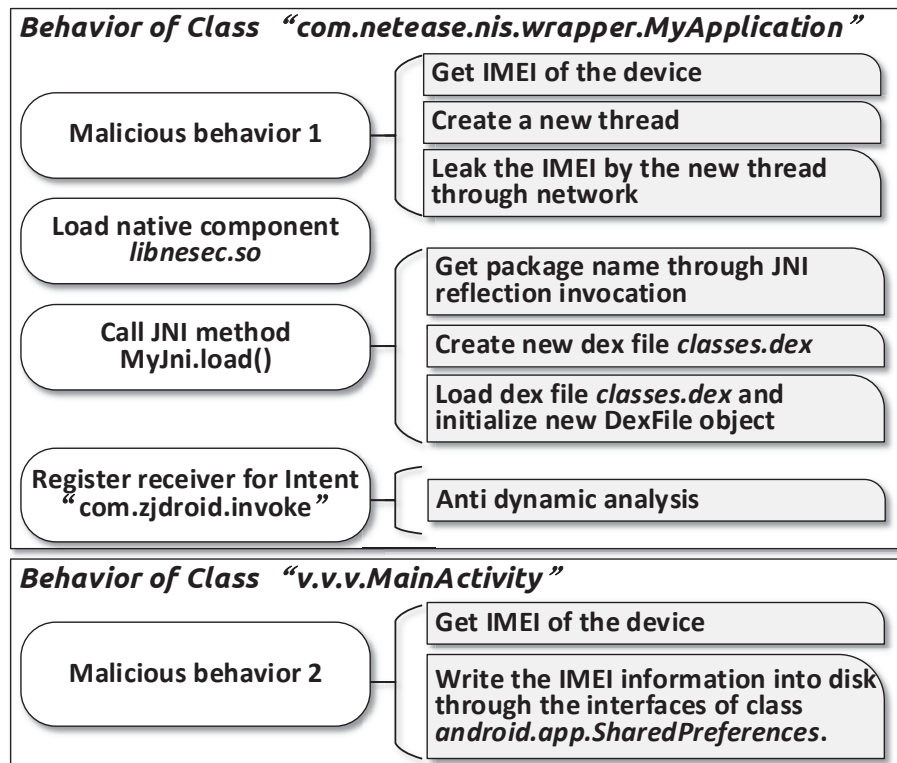


Figure 3.7: The behaviours reconstructed by Malton.

into the file “.cache/classes.dex” under the app’s directory (Line 13 and 14). After that, a new `DexFile` object is initialized based on the newly created Dex file through the runtime function `DexFile::OpenMemory()` (Line 16).

We also find that the packed malware registers an Intent receiver to handle the Intent `com.zjdroid.invoke` at Line 19 and 21. Note that ZjDroid [32] is a dynamic unpacking tool based on the Xposed framework and is started by the Intent `com.zjdroid.invoke`. By registering the Intent receiver, the malware can detect the existence of ZjDroid.

Finally, the app loads and initializes the class `v.v.v.MainActivity` in Line 23 to 26, and the hidden malicious payloads are executed at Line 29. To hide itself, the malware also calls the framework method `Activity.finish()` to destroy its activity (Line 30).

Summary Malton can analyze sophisticated packed malware samples, and help the analyst identify the behaviours of both the malware and its hidden code.

3.4.3 Path Exploration

To answer **Q3.3**, we first employ Malton to analyze the SMS handler of the packed malware *com.nai.ke*. From the logs, we find that its SMS handler *handleReceiver()* processes each incoming SMS by obtaining its address and content through methods *getOriginatingAddress()* and *getMessageBody()*, respectively. If the SMS is not from the controller (i.e., Tel: 1851130**14), it calls the method *abortBroadcast()* to abort the current broadcast.

Effectiveness To explore all the malicious payloads controlled by the received SMS message, we specify the code region between the return of the function *getMessageBody()* and the return of *handleReceiver()* to perform in-memory concolic execution. We set the result of *getMessageBody()* (i.e., SMS content) as the input of the concolic execution. To circumvent the checking of the phone number of the received SMS message, we trigger the malware to execute the satisfied code path by changing the result of *getOriginatingAddress()* to the number of the controller.

However, we find that the constraint resolver cannot always find the satisfying input due to the comparison between two strings' hash values. Therefore, we use the directed execution engine to force the malware to execute the selected code path. Eventually, we identify 14 different code paths (or behaviours) that depend on the content of the received SMS. The generated inputs and their corresponding behaviours are listed in Table 3.5. This result demonstrated the effectiveness of Malton in exploring different code paths.

Efficiency Thanks to the in-memory optimization, when exploring code paths in the

Table 3.5: The commands and related behaviours explored by Malton (The 3rd column represents the number of IR blocks, required to execute for exploring the behaviour of the corresponding command, with/without in-memory optimization).

Command	Detected behaviour	Number of executed blocks
“cq”	Read information SMS contents, contacts, device model and system version, then send to 292019159e@fcv77f.com with password “aAaccv11” through SMTP protocol.	32k/20443k
“qf”	Send SMS to all contacts with no SMS content.	7k/20537k
“df”	Send SMS to specified number, and both the number and content are specified by the command SMS.	5k/22970k
“zy”	Set unconditional call forwarding through making call to “**21*targetNum%23”, and the <i>targetNum</i> is read from the command SMS.	8k/22848k
“by”	Set call forwarding when the phone is busy through making call to “%23%23targetNum%23”, and the <i>targetNum</i> is read from the control SMS.	15k/20639k
“ld”, “fd”, “dh”, “cz”, “fx”, “sx”, “dc”, “bc”	Modify the its configuration file zzzx.xml.	5k-18k/20403k-20452k
Others	Tell the controller the command format is error by replying a SMS.	15k/20443k

Table 3.6: The number of IR blocks executed for path exploration with and without in-memory optimization.

Malware	With Optimization	Without Optimization
<i>0710ef0ee60e1acfd2817988672bf01b</i>	203k	26237k
<i>0ced776e0f18ddf02785704a72f97aac</i>	203k	26010k
<i>0e69af88dcb469e30f16609b10c926c</i>	4k	16826k
<i>336602990b176cf381d288b79680e4f6</i>	13k	1908k
<i>8e1c7909aed92eea89f6a14e0f41503d</i>	7k	69968k

interested code region, Malton just needs one SMS and then iteratively executes the specified code region for 14 times without the need of restarting the app for 14 times. To evaluate the efficiency of the in-memory optimization, we record the number of IR blocks to be executed for exploring each code path with/without in-memory optimization, and list them in Table 3.5 (the last column). The result shows that the in-memory optimization can avoid executing a large number of IR blocks. For example, when exploring the paths decided by the command “df”, Malton only needs to execute 5k IR blocks with in-memory optimization. Otherwise, it has to execute 22,970k IR blocks.

We also use another five malware samples, which have the SMS handler, to further evaluate the efficiency of path explorer module. The average number of IR blocks to be executed with and without in-memory optimization are listed in Table 3.6. The in-memory optimization can obviously reduced the number of IR blocks to be executed.

Summary The path exploration module of Malton can explore code paths of malicious payloads effectively and efficiently. The concolic execution engines generates the satisfying inputs to execute certain code paths, and the directed execution engine forcibly executes interested code paths when constraint resolver fails.

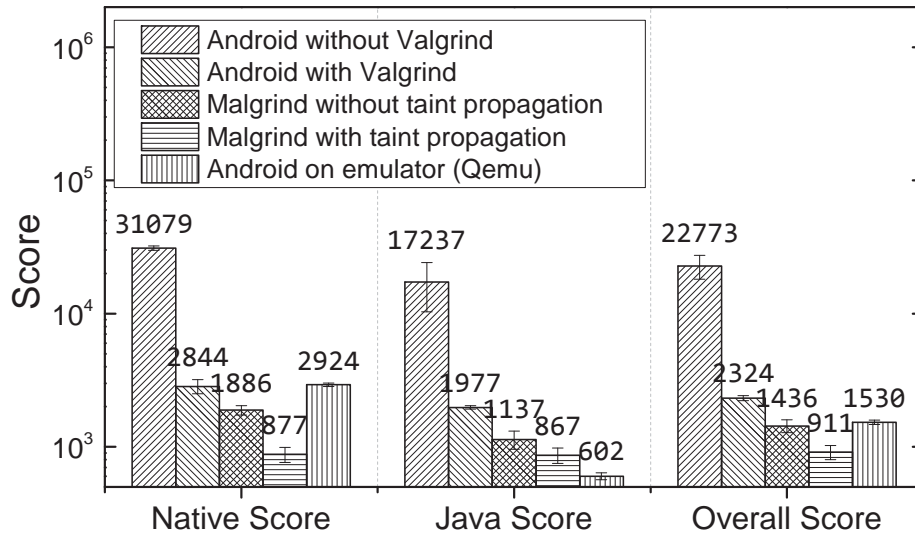


Figure 3.8: Performance measured by CF-Bench.

3.4.4 Performance Overhead

To understand the overhead introduced by Malton, we run the benchmark tool CF-Bench [25] 30 times on the smartphone (Nexus 5 running Android 6.0) under four different environments, i.e., Android without Valgrind, Android with Valgrind, and Malton with and without the taint propagation. Moreover, to compare with the dynamic analysis tools based on Qemu, we also run CF-Bench inside the Qemu emulator, which runs on a Ubuntu 14.04 desktop equipped with Core(TM) i7 CPU and 32G memory.

The results are shown in Figure 3.8. There are three types of scores. The Java score denotes the performance of Java operations, and the native score indicates the performance of naive code operation. The overall scores is calculated based on the Java and the native score. A higher score means a better performance.

From the Figure, we can find that Malton introduces around 16x and 36x slowdown to the Java operations without and with taint propagation. However even if the app runs with only Valgrind, there is 11x slowdown. That means Malton brings 1.5x-3.2x additional

slowdown to Valgrind. Similarly, for the native operations, Malton introduces 1.7x-2.3x additional slowdown when Valgrind is taken as the baseline. Overall, Malton introduces around 25x slowdown (with taint propagation) to the app to be analyzed. Moreover, because the Qemu [20] incurs around 15x overall slowdown, and the Qemu-based tools, such as the taint tracker of DroidScope [180], usually incurs another 11x-34x additional slowdown, Malton is more efficient compared with the existing tools based on the Qemu emulator.

Summary As a dynamic analysis tool, Malton has a reasonable performance, and it is more efficient than the existing tools based on the Qemu emulator.

Chapter 4

Adaptive Unpacking of Android Apps

4.1 Overview

More and more app developers use the packing services (or packers) to prevent attackers from reverse engineering and modifying the executable (or Dex files) of their apps. At the same time, malware authors also use the packers to hide the malicious component and evade the signature-based detection. Although there are a few recent studies on unpacking Android apps, it has been shown that the evolving packers can easily circumvent them because they are not adaptive to the changes of packers. In this thesis, we propose a novel adaptive approach and develop a new system, named *PackerGrind*, to unpack Android apps. We also evaluate *PackerGrind* with real packed apps, and the results show that *PackerGrind* can successfully reveal the packers' protection mechanisms and recover the Dex files with low overhead, showing that our approach can effectively handle the evolution of packers.

In summary, our major contributions include:

- We propose a new iterative process to unpack Android apps. This process as well as the new system, named *PackerGrind*, is adaptive to the evolution of packers.

- We design *PackerGrind* that automates most steps in the iterative process. It can conduct cross-layer monitoring and Dex file recovering in real smartphones. Moreover, it supports both DVM and ART. To our best knowledge, it is the *first* system that can address the above two challenging research questions simultaneously.
- We implement *PackerGrind* with 21.3K lines of C/C++ code (not include Valgrind) and 2.5K lines of Python code, and compare it with the state-of-the-art unpacking tools with real apps packed by popular packers. The results show that *PackerGrind* can unpack all these apps with low overhead whereas DexHunter[193] recovered a few and Android-unpacker[2] unpacked none.

The rest of this chapter is organized as follows. Chapter 4.2 introduces background knowledge and a motivating example. Chapter 4.3 describes the basic Dex data collection points. Chapter 4.4 details the design and implementation of *PackerGrind* and Chapter 4.5 reports the experimental results.

4.2 Background

4.2.1 Dex File

The bytecode of an Android app is contained in the Dex file which is a highly structured data file consisting of different Dex data items [26](e.g., `proto_id_item`, `code_data_item`). A Dex file has three major sections including *header* section, *data identifiers* section, and *data* section. The *header* section includes a summary of the Dex file (e.g., checksum, size, and offsets). The *data identifiers* section contains 6 identification lists for defined classes, namely, `string_ids`, `type_ids`, `proto_ids`, `field_ids`, `method_ids`, and `class_defs`, each of which contains multiple items. For example, `string_id_item` contains the offset from the start of the Dex file to the corresponding

`string_data_item`. The *data* section contains the information related to bytecode, including `map_list`, `type_list`, `class_data_items`, `code_data_items`, `debug_info_items`, `encoded_array_items`, and four annotation data items.

4.2.2 Android App Packing

Packers usually protect apps' code from three aspects, namely, hiding Dex files, impeding the dumping of Dex files in memory, and hindering the reverse-engineering of Dex files.

Hiding Dex files. Packers often use three approaches to hide Dex files. (1) *Dex file modification*. Packed apps use native code to modify Dex files in the memory when the app is running. For example, apps packed by Baidu packer in 2015 fill a special method with valid instructions just before the method is called and erase them after execution. *PackerGrind* can capture such behaviours and dump the correct instructions at the right moment. (2) *Dynamic class loading*. Packers put the bytecode of selected functions in separated Dex files, and load them when the functions are invoked. They even encrypt the Dex files and decrypt them before loading the required classes. *PackerGrind* can dump the Dex files after they are loaded because it traces the runtime's functions. (3) *Native method*. Packers could turn the selected Dex functions into native methods and then invoke them through Java native interface (JNI) from the Dex file. Although *PackerGrind* is not designed to reverse engineer the native code for regenerating the bytecode, it can still provide useful information about the native methods thanks to its cross-layer monitoring component.

Impeding the dumping of Dex files. Packers usually employ three approaches to prevent unpackers from dumping the real code in memory. (1) *Emulator detection*. Since many dynamic analysis systems rely on Android emulator, packers employ advanced techniques [166] to determine whether a packed app is running in an emulator. If so,

the app will exit. *PackerGrind* does *not* rely on the emulator. Instead, it exploits dynamic binary translation [126] to perform cross-layer monitoring and recovers Dex files.

(2) *Anti-debug*. If an unpacker attaches to the packed apps as a debugger, it can monitor the apps and obtain the Dex files. To impede such method, the packed apps often launch multiple threads and let one thread attach to another using *ptrace*, because a process can only be attached by one process. *PackerGrind* does *not* use this approach.

(3) *Hooking*. To prevent unpackers from accessing and dumping the Dex files in memory, packed apps often hook the functions related to file and memory operations to prohibit unpackers from using them. *PackerGrind* can disable these hooks.

Anti reverse-engineering of Dex files. Packers commonly employ various techniques (e.g., obfuscation[70], etc.) to raise the bar for understanding the internal logics through static code analysis. Handling them is out of the scope of *PackerGrind*.

4.2.3 A Motivating Example

Existing unpackers are not adaptive to the changes of packers, and hence can be easily circumvented. In particular, they usually perform one-pass processing based on the developer's knowledge for obtaining the Dex files. Therefore, packers can modify their behaviours accordingly to defeat such unpackers. We argue that unpackers should be adaptive to the changes of packers by monitoring and learn their behaviours.

We use an app packed by Baidu packer (in DB-15) [55] as a motivating example. As shown in Figure 4.1, the original code of *onCreate()* in MainActivity is replaced by those at Line 23-26, and *onCreate001()* is empty and called between two JNI methods (i.e., *A.d()* and *A.e()*). By monitoring the packed app, we find that when *A.d()* is invoked, it fills *onCreate001()* with correct instructions, which will be erased after *A.e()* is called.


```

22 public void onCreate(Bundle bundle) {
23     String str = "LXXX;->onCreate001(Landroid/os/Bundle;)V";
24     A.d(str);
25     onCreate001(bundle);
26     A.e(str);
27 }
28 private void onCreate001(Bundle savedInstanceState) {
29 }

```

Figure 4.1: *onCreate()* method of the app packed by the Baidu packer of DB-15.

The state-of-the-art unpackers (i.e., DexHunter [193] and AppSpear [182]) cannot obtain the instructions in *onCreate001()* effectively. DexHunter collects the Dex data in *dvmDefineClass()*. However, when this function is called, the correct instructions have not been written to *onCreate001()* yet, and thus DexHunter misses them. Similarly, AppSpear assumes that DVM’s parsing methods (e.g., *dexGetCode()*) always provide expected results. However, in this example, *dexGetCode()* can only obtain the right instructions of *onCreate001()* when it is invoked between *A.d()* and *A.e()*. In other words, AppSpear cannot get the correct results if it misses the right moment. Since DexHunter and AppSpear are implemented within the runtime, they cannot monitor packed apps’ behaviours at either the system or the instruction levels to determine the right moments.

PackerGrind can address this issue because it iteratively monitors packed apps at different layers, facilitates the determination of collection points, and recovers the Dex files. By analyzing the Dex file obtained in the first run, we can learn that the instructions of *onCreate001()* are modified during the execution. According to the tracking report, we know that *A.d()* is called before *onCreate001()* to fill the instructions and *A.e()* is invoked after *onCreate001()* to erase them. Moreover, as shown in Figure 4.1, the parameters of both method *A.d()* and *A.e()* are the name of *onCreate001()*. With such information, we add a new collection point between *A.d()* and *A.e()*, and then *PackerGrind* can reconstruct the correct Dex file automatically.

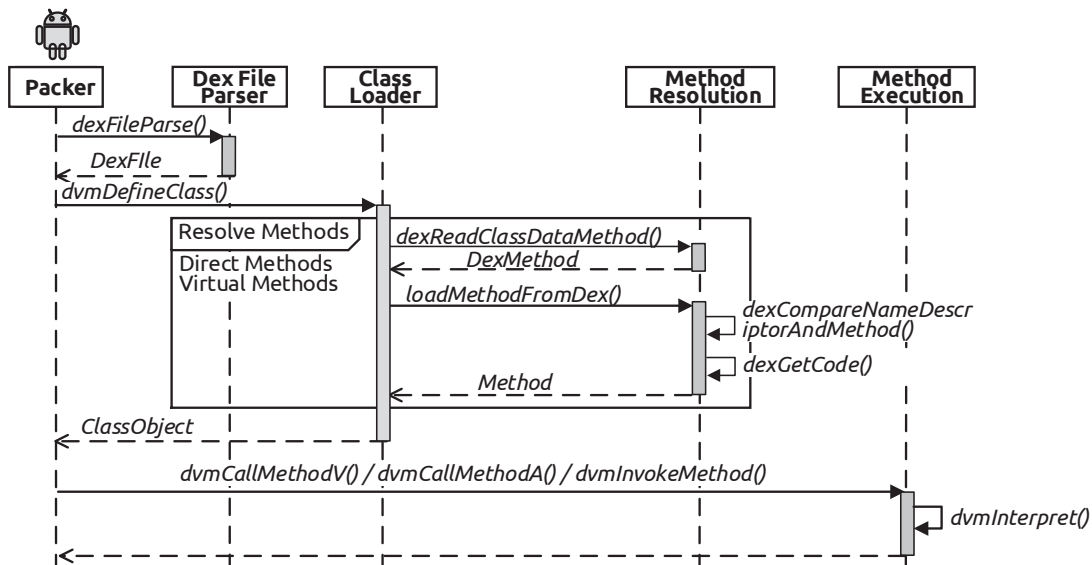


Figure 4.2: The process from Dex file loading to method execution.

4.3 Basic Dex Data Collection Points

As shown in Figure 4.2, we divide the process from Dex files loading to method execution into four phrases, namely, parsing Dex files, loading classes, resolving methods, and executing methods. Consequently, we define four basic collection points for DVM and ART, respectively.

4.3.1 Dalvik VM (DVM)

Parsing Dex Files. A Dex file can be loaded either from a file in storage through `openDexFileNative()` or a memory space through `openDexFile_bytearray()`. Both methods will call `dexFileParse()` to parse the Dex file and return the structure `DexFile` to represent this Dex file at runtime, as shown in Figure 4.2. Since `DexFile` is initialized according to the Dex file header in `dexFileParse()`, we select `dexFileParse()` as the first Dex data collection point.

Loading Classes. A class can be loaded through `Dalvik_dalvik_system_DexFile_defineClassNative()`. In this function, `dvmDefineClass()` is called to load the class and return the structure `ClassObject` that contains the class's information (e.g., fields, methods, etc.). Moreover, the structure `class_def_item` is read from the Dex file, and then the structure `class_data_item` is parsed from the Dex file according to its offset in `class_def_item`. After that, `ClassObject` is initialized. Hence, we choose `dvmDefineClass()` as the second Dex data collection point.

Resolving Methods. When loading a class, the class loader will resolve each method to initialize `ClassObject` according to `class_data_item`. During such resolution, the class loader first obtains `DexMethod` from the Dex file by calling `dexReadClassDataMethod()`. Then, it creates a structure `Method` according to `DexMethod` in `LoadMethodFromDex()`. During the initialization of `Method`, `dexCompareNameDescriptorAndMethod()` is called to check whether it is a *finalize* method, and then `dexGetCode()` is invoked to fetch the code information from the Dex file to populate `Method`. Since the symbols of inline functions and static functions are not exported in `libdvm.so`, we choose `dexCompareNameDescriptorAndMethod()` rather than `dexGetCode()` as the third Dex data collection point.

Executing Methods. Native code can invoke Java methods through Java reflection or JNI reflection using functions like `dvmInvokeMethod()`, `dvmCallMethodA()`, and `dvmCallMethodV()`. Since they call `dvmInterpret()` for both fast-interpreter and portable-interpreter, we select it as the fourth Dex data collection point.

4.3.2 Android Runtime (ART)

During the installation of an app, ART invokes the tool `dex2oat` to compile the Dex file to the oat file, which is in ELF format but contains both the Dalvik bytecode and

the compiled code. If an app without oat file is being launched, ART performs the same action. ART can execute a method in the interpreter mode, which is similar to DVM, or the compiled code mode. By default, if a method has compiled code, ART runs its compiled code. Otherwise, ART interprets its Dalvik bytecode. If a packed app uses `dex2oat` to compile Dex files containing real code into oat file, *PackerGrind* obtains the Dex file according to the arguments passed to `dex2oat`.

For the methods executed in the interpreter mode, *PackerGrind* also has four basic Dex data collection points.

Parsing Dex Files. Similar to DVM, `DexFile` represents the Dex file in runtime, which contains the information of classes and methods. The class constructor of `DexFile` (i.e., *DexFile()*) will read the Dex file in memory and parse it similarly to *dexFileParse()* in DVM. Therefore, we choose *DexFile()* as the first Dex data collection point.

Loading Classes. *DefineClass()* of class `ClassLinker` is used to load and parse each class in the Dex file and return an instance of class `Class` to represent the class in runtime. Hence, we select *DefineClass()* as the second Dex data collection point.

Resolving Methods. ART uses `ArtMethod` to represent each method of a class, and the instance of `ArtMethod` for a method is initialized in *LoadMethod()*. Therefore, we take it as the third Dex data collection point.

Executing Methods. *Invoke()* of the class `ArtMethod` in ART is invoked when a Java method is called by Java reflection and JNI reflection. Hence, we select *Invoke()* as the fourth Dex data collection point.

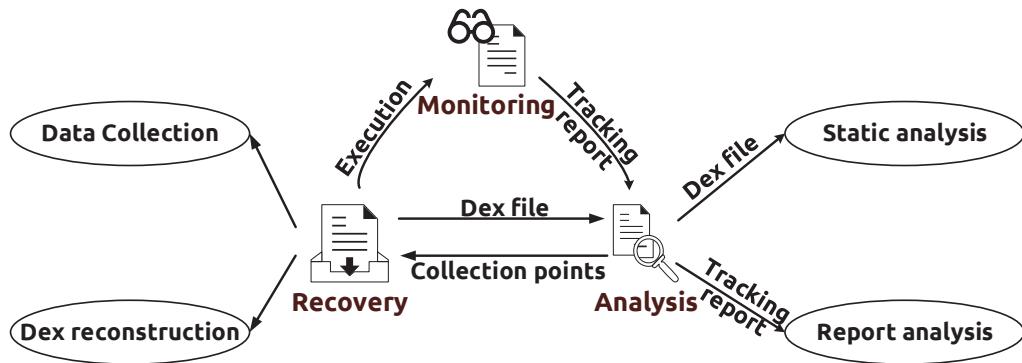


Figure 4.3: The iterative process realized by *PackerGrind*.

4.4 PackerGrind

4.4.1 Overview

To be adaptive to the evolution of packers, *PackerGrind* adopts the iterative process shown in Figure 4.3 to recover Dex files. This process consists of three tasks in each run. More precisely, when running a packed app in smartphone, *PackerGrind* monitors its behaviours from three layers including runtime, system, and instruction, and generates a tracking report. At the same time, *PackerGrind* collects Dex data at specified collection points and reconstructs Dex files by the end of each run. Then, it performs static analysis on the recovered Dex files. Users determine whether new collection points are needed according to the tracking report and the result of static analysis, because the basic data collection points described in Chapter 4.3 may not be enough for *PackerGrind* to collect all data of the original Dex file. We propose basic protection patterns in Chapter 4.4.5 to help users determine additional collection points if needed. Based on these patterns, we have identified all collection points for packers accessible to us as described in Chapter 4.5. After adding the new collection points, *PackerGrind* will run the process one more time and repeat this procedure until the Dex file is correctly recovered.

Figure 4.4 shows the architecture of *PackerGrind*. It consists of three components

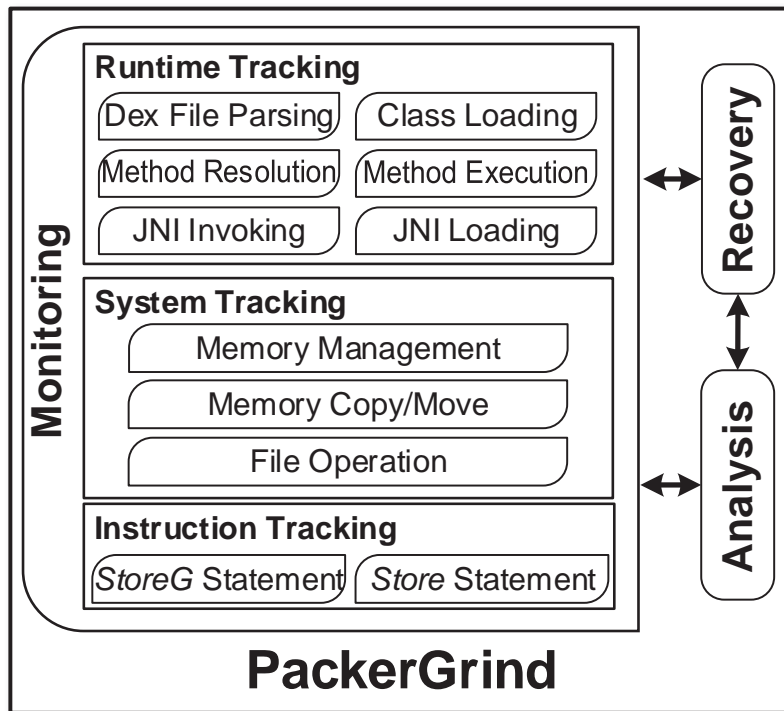


Figure 4.4: Architecture of *PackerGrind*.

for finishing the three tasks in the iterative process. The monitoring component (Chapter 4.4.2) tracks the packed apps at three layers and generates the tracking report. The recovery component (Chapter 4.4.4) automatically gathers Dex data at selected collection points and reconstructs the Dex file. The analysis component (Chapter 4.4.5) performs static analysis on the Dex file dumped at each run and determines whether new data collection points are needed. We develop *PackerGrind* based on Valgrind [126] and therefore it runs in a real smartphone instead of an emulator.

4.4.2 Monitoring

Runtime layer To locate the structure `DexFile`, which represents a Dex file in runtime, and collect Dex data from the four basic data collection points (Chapter 4.3), *PackerGrind* monitors the arguments and the returns of the selected functions in Table 4.1 using

Table 4.1: Wrappers for tracking Dex and DVM related events.

Category	Wrapped Functions	Tracked Information
Dex Data	<i>dexFileParse()</i>	Dex file parsing
	<i>dvmClassDefine()</i>	Class loading
	<i>dexCompareNameDescriptorAndMethod()</i>	Method resolution
	<i>dvmInterpret()</i> , <i>dvmMterpStdRun()</i>	Method execution
	<i>dvmCallJNIMethod()</i>	JNI invocation
	<i>dvmInvokeMethod()</i>	Java reflection
Native Module	<i>dvmCallMethodV()</i> , <i>dvmCallMethodA()</i>	JNI reflection
	<i>dvmLoadNativeCode()</i>	Native code loading

the function wrapping technique[126]. For example, by wrapping *dexFileParse()* with the wrapper function *dexFileParse_wrapper()*, we can obtain the arguments passed to *dexFileParse()* and its return.

Table 4.1 lists two set of functions. One includes the functions related to the basic Dex data collection points and the function *dvmCallJNIMethod()*, because some packers use native code to modify Java methods through JNI. Moreover, it contains the functions related to Java reflection and JNI reflection (i.e., *dvmInvokeMethod()*, *dvmCallMethodV()*, *dvmCallMethodA()*), because they are used by some packers to invoke Java methods. The other set has *dvmLoadNativeCode()* because it will be called when *System.load()* or *System.loadLibrary()* is used to load native module. Since native modules allow packed apps to release or modify the Dex data, we wrap *dvmLoadNativeCode()* to track such behaviours.

System layer Packed apps can release and modify the Dex data in memory by calling system library functions and system calls through its native module. Since such behaviours cannot be monitored at the runtime layer, *PackerGrind* tracks them at the system layer by wrapping memory management functions (e.g., allocation, free, mapping, etc.), file operations (e.g., open, read, write, and close), and data movement functions (e.g.,

memcpy(), *strcpy()*, etc.). It also traces the invocation of some system calls (e.g., *sys_map()*, *sys_unmap()* and *sys_protect()*), because they can be used by packed apps to allocate memory, release memory, and change memory access permissions, respectively. *PackerGrind* maintains a surveillance memory list for the ranges of memory that may be used to store the Dex data, and records the operations on them in tracking report.

PackerGrind also wraps some functions for special purposes. For example, some packers adopt timeout mechanism for anti-unpacking (e.g., Ijiami). More precisely, if the unpacking process takes a time longer than the packer's timeout threshold, the app crashes. To address this issue, we wrap the system call *sys_gettimeofday()* to modify the timestamps returned to the packer so that its timeout mechanism will not be activated. Users can use *PackerGrind* to track more functions if necessary.

Instruction layer *PackerGrind* instruments *store* instructions to monitor operations for modifying Dex files, because packed apps can write and modify Dex data in memory directly through its native code instead of invoking memory copy or move functions. *PackerGrind* skips system libraries, because no system library functions except memory copy and move functions, which are wrapped at the system layer, will modify Dex files. *PackerGrind* maintains a system library memory list for the memory regions of system libraries, and uses it to determine whether an instruction belongs to system libraries.

To monitor memory modifications, *PackerGrind* inserts an intermediate representation (IR) of function invocation statement before `Ist_StoreG` and `Ist_Store` statements that are translated from packed apps' native code by Valgrind [126]. In this IR statement, the instruction tracking function will be called to check whether the target address is in the surveillance memory list. If so, *PackerGrind* records the target address, operand value, and instruction address in the tracking report.

4.4.3 Tracking report

A tracking report contains three major types of information and its length depends on the app's execution time.

(1) Dex file. When a new Dex file represented by `DexFile` is found, *PackerGrind* parses `DexFile` and records the memory information about the Dex file (e.g., Dex file header, classes, methods and codes).

(2) Memory modification. *PackerGrind* maintains a Dex file list containing the memory ranges of all Dex files in the runtime. When functions and instructions for memory modification are identified, *PackerGrind* checks whether the target addresses are in the memory range of a Dex file. If so, the modification information is written to the tracking report. At the system layer, this information includes the invoked function, target address, the Dex structure to which the target address belongs (e.g., Dex header field), and the value written to the target address. At the instruction layer, this information includes instruction address, instruction types (i.e., `Ist_StoreG` and `Ist_Store`), target address, target address information, and the stored value.

(3) Method invocation. At the runtime layer and the system layer, the invocation and the return of any wrapped function are logged into the tracking report with the parameters and the return values.

4.4.4 Recovery

It collects the Dex data and reconstructs Dex files.

Dex data collection. In each run, *PackerGrind* starts collecting Dex data after a `DexFile` is identified because it represents a Dex file. Once the Dex file is located through

```

001 Syscall: open() /data/dalvik-cache/data@app@demo.killerud.gestures-1.apk@classes.dex Flag=0x00020000 fd=50
002 .....
003 Invoke: dvmLoadNativeCode() /data/app-lib/demo.killerud.gestures-1/libmobisec.so // First dex file is opened and tracking starts
004 Return: dvmLoadNativeCode() /data/app-lib/demo.killerud.gestures-1/libmobisec.so // Load the native library named libmobisec.so
005 Invoke: dvmCallJNIMethod() pDexFile=0x05f3ef40 mth: Lcom/all/mobisecenhance/StubApplication; attachBaseContext(TTLVL) // JNI method attachBaseContext(TTLVL) is invoked
006 Syscall: open() 0x61c6faf0/data/data/demo.killerud.gestures/files/libmobisecx1.so Flag=0x00020042 fd=54 // File named libmobisecx1.so is opened and file handler is 54
007 Syscall: mmap() off_0x00000000 -> 0x375b6000-0x375b7a58 flak=rw prot=0x2 fd=54 // File libmobisec1.so is mapped to memory range 0x375b6000-0x375b7a58
008 Invoke: dexFileParse() file: 0x375b6000-0x375b7a58 flag: KdexParseDefault pDexFile=0x00000000 // dexFileParse() is invoked to parse memory range 0x375b6000-0x375b7a58
009 Return: dexFileParse() file: 0x375b6000-0x375b7a58 flag: KdexParseDefault pDexFile=0x05f44970 // dexFileParse() returns results: pDexFile=0x05f44970
010 Syscall: close() fd=54 // File libmobisec1.so is closed
011 3759DC87: Executable | STORE *a_0x375b654c <- v_0x24 | InterFacesOff (pDexFile=0x05f44970 ClassIdx=5) // The interfaceOff (ClassIdx=5) value is modified by native code
012 3759E1F3: Executable | STORE *a_0x375b6b2a <- v_0x80 | class_data_item (pDexFile=0x05f44970 ClassIdx=0) // The code_data_item of the class (ClassIdx=0) is modified by native code
013 3759E1F9: Executable | STORE *a_0x375b6b2b <- v_0x80 | class_data_item (pDexFile=0x05f44970 ClassIdx=0) // The code_data_item of the class (ClassIdx=0) is modified by native code
014 .....
015 3759E205: Executable | STORE *a_0x375b6bc7 <- v_0x80 | class_data_item (pDexFile=0x05f44970 ClassIdx=9) // The code_data_item of the class (ClassIdx=9) is modified by native code
016 3759E209: Executable | STORE *a_0x375b6bc8 <- v_0x00 | class_data_item (pDexFile=0x05f44970 ClassIdx=9) // The code_data_item of the class (ClassIdx=9) is modified by native code
017 Return: dvmCallJNIMethod() pDexFile=0x05f3ef40 mth: Lcom/all/mobisecenhance/StubApplication; attachBaseContext(TTLVL) // JNI method attachBaseContext(TTLVL) returns
018 Invoke: dvmDefineClass() pDexFile=0x05f44970 class: Lhiof/enigma/android/gestures/GesturesDemoActivity; // dvmDefineClass() is invoked to define class GesturesDemoActivity;
019 Return: dvmDefineClass() pDexFile=0x05f44970 class: Lhiof/enigma/android/gestures/GesturesDemoActivity; // dvmDefineClass() returns.

```

Figure 4.5: Tracking report for an app packed by the Ali packer.

DexFile, *PackerGrind* initializes a shadow memory for storing the collected Dex data items belonging to this Dex file, and then copies the data items to the shadow memory. When a new Dex data item is collected, *PackerGrind* firstly checks whether the shadow memory for this data item exists. If it does, *PackerGrind* copies this data item to the shadow memory. Otherwise, *PackerGrind* creates a new shadow memory for this data item, copies it to the shadow memory, and changes the corresponding offset to this item in the shadow memory.

Dex file assembling. After collecting Dex data, *PackerGrind* assembles them into a Dex file. Since a packer can release Dex data in discontinuous memory areas, there will be more than one shadow memory allocated for storing the collected Dex data. Therefore, *PackerGrind* allocates a continuous memory and assembles the collected Dex data together to reconstruct Dex files. Specifically, *PackerGrind* performs a two-step Dex file construction. First, it divides the collected Dex data items into different groups according to their types. For example, *PackerGrind* groups all `class_def_items` together to assemble `class_defs`. After that, these groups of data will be put together according to the Dex format. By doing so, *PackerGrind* can obtain the offsets of the Dex data items and the sizes needed for such data structures.

Second, *PackerGrind* allocates a continuous memory region and copies the collected data to it starting from their group offsets. For each data structure, *PackerGrind* updates its members according to the offsets of the data structures. For example, when a `class_data_item` is copied into the continuous memory region, we will update the corresponding `class_def_item.class_data_off`. *PackerGrind* will recalculate the meta-data of Dex header after all data structures are copied into the continuous memory region to ensure the validity of Dex file. Eventually, *PackerGrind* dumps the memory region and outputs the Dex file.

4.4.5 Analysis

We analyze the dumped Dex file and the tracking report to achieve three purposes. First, since packed apps usually use JNI methods (i.e., native code) to dynamically modify Dex files in memory and the dumped Dex file may contain unexplored paths to JNI methods, we conduct static bytecode analysis to look for such paths. Second, we inspect the tracking report to determine whether new data collection points are needed. Third, we identify more information about the discovered JNI methods from the result of cross-layer monitoring.

Static bytecode analysis. We employ IntelliDroid [172] to determine how to trigger the JNI methods in the dumped Dex files through static analysis. Given an app and a set of targeted JNI methods, IntelliDroid [172] can help us find the execution paths leading to these methods as well as the corresponding input. Thus, we first extract JNI methods from the Dex files and let them be the target methods, and then use IntelliDroid to look for the execution paths leading to them with event handlers as the entry-points. After that, we drive the app to execute the target JNI methods following the corresponding paths.

Tracking report analysis. We provide Python scripts to analyze the tracking report in order to recognize the protection patterns and determine whether new collection points are needed. By exploiting the insight that a portion P (e.g., Dex header, methods, etc.) of a Dex file should be valid right before it is being used, we define four basic protection patterns for P : (1) it is changed to valid value before its first use (**FmT**); (2) it is modified to invalid value after its last use (**TmF**); (3) it is altered to valid value before being used and turned to invalid after the use (**FmTmF**); (4) it is always valid (**T**). Although the basic protection patterns are by no means comprehensive, they cover all packed samples accessible to us. Users can define new patterns after studying the tracking report.

To recognize the protection patterns, we first collect the method invocation and the

memory modification information from the tracking report. According to the target address information of each modification operation M , we identify which portion of the Dex file is modified by M . By checking the method invocation information, we determine when the portion is used. Given each portion P , we regard its content as valid when it is being used. A quick approach to detect invalid portions is applying static analysis to the dumped Dex file and see whether there is any parsing error.

During the first run, *PackerGrind* collects Dex data at the basic data collection points, and hence the dumped Dex file may include invalid portion. If so, we infer the protection pattern and select new data collection points (i.e., when its content is valid). After that, we execute *PackerGrind* again to collect more valid portions.

We use an app packed by Ali packer as an example to illustrate this process. Figure 4.5 shows the tracking report. At line 003, *dvmLoadNativeCode()* is called to load the native library `libmobisec.so`. Then, the JNI method *attachBaseContextIT(VL)* of class *Lcom/ali/mobisecenhance/StubApplication* is called by *dvmCallJNIMethod()* (line 005), and it returns at line 017.

The information about the instruction layer modifications is from line 011 to line 016. At line 011, “3759DC87” and “Executable” are the instruction address and the executable permission of the address, respectively. “STORE” indicates the instruction type, which is `Ist_Store` at line 011. “a_0x375b654c” and “v_0x24” are the target address and the stored value, respectively. “*interfacesOff* (pDexFile=0x05f44970 ClassIdx=5)” denotes that the target address is the field *interfacesOff* of the 5th class in the Dex file, which is represented by a `Dexfile` in memory address `0x05f44970`. At line 11, the field *interfaceOff* of the 5th classes (*ClassIdx=5*) in the Dex file is modified. From line 012 to 016, the `class_data_items` of 10 classes (from *ClassIdx=0* to *ClassIdx=9*) in the Dex file are modified by the STORE (i.e., `Ist_Store`) instruction. Finally, the class

```

1 public void onCreate(Bundle savedInstanceState) {
2     super.onCreate(savedInstanceState);
3     setContentView(C0000R.layout.main);
4     this.display = ((WindowManager) getSystemService("window")).getDefaultDisplay();
5     this.mLibrary = GestureLibraries.fromRawResource(this, C0000R.raw.gestures);
6     if (!this.mLibrary.load()) {
7         finish();
8     }
9     findViewById(C0000R.id.gestures).addOnGesturePerformedListener(this);
10 }

```

(a) The original *onCreate()*.

```

1 public void onCreate(Bundle savedInstanceState) {
2     A.V(0, this, new Object[]{savedInstanceState});
3 }

```

(b) The *onCreate()* in a packed app.

```

001 Invoke:dvmCallJNIMethod() pDexFile=0x05f41a90 mth: Lcom/baidu/protect/A; V(VILL)
002     JNI_Reflection: Landroid/app/Activity; onCreate(VL)
003     JNI_Reflection: Landroid/app/Activity; setContentView(VI)
004     JNI_Reflection: Landroid/app/Activity; getSystemService(LL)
005     JNI_Reflection: Landroid/view/WindowManagerImpl; getDefaultDisplay(L)
006     JNI_Reflection: Landroid/gesture/GestureLibraries; fromRawResource(LLI)
007     JNI_Reflection: Landroid/gesture/GestureLibraries$ResourceGestureLibrary; load(Z)
008     JNI_Reflection: Landroid/app/Activity; findViewById(LI)
009     JNI_Reflection: Landroid/gesture/GestureOverlayView; addOnGesturePerformedListener(VL)
010 Return:dvmCallJNIMethod() pDexFile=0x05f41a90 mth: Lcom/baidu/protect/A; V(VILL)

```

(c) Tracking report of *A.V()*.

Figure 4.6: The method *onCreate()* before and after packing and the tracking report of *A.V()*.

GesturesDemoActivity is defined by *dvmDefineClass()* at line 018 and 019.

The tracking report shows that the `class_data.items` of all classes are filled with valid values in the JNI method *attachBaseContextIT(VL)* before *dvmDefineClass()*. Since the runtime loads classes in *dvmDefineClass()* based on `class_data.items`, the `class_data.items`' contents are valid after calling *attachBaseContextIT(VL)*. That means, `class_data_item` follows the **FmT** protection pattern. Hence, we can collect the Dex data after *attachBaseContextIT(VL)* returns. Since *PackerGrind* collects Dex data when it is defined by *dvmDefineClass()* by default, the collected content is valid and no more run is needed.

Native method inspection. A packer can re-implement an app's Java methods in the native module, and then call them through JNI. Although *PackerGrind* is not designed to reverse-engineer the native code for reconstructing the bytecode, it can still provide useful information about the native methods thanks to its cross-layer monitoring capability. More precisely, packed apps have to use JNI reflection (i.e., *dvmCallMethodV()* or *dvmCallMethodA()*) to invoke Android framework APIs in Java. Since *PackerGrind* has wrapped these functions, it can provide rich information about the native methods.

For example, Figure 4.6 shows the method *onCreate()* in the original app and that in the app packed by Baidu. It shows that the method *onCreate()* has been re-implemented in native code, which invokes Android framework APIs through JNI reflection. In other words, after packing, the original implementation of *onCreate()* is replaced with the invocation of the native method *A.V()*. From the tracking report of *A.V()* shown in Figure 4.6(c), we can infer the original implementation of *onCreate()*. For example, the method *onCreate()* of class *android/app/Activity* is invoked at Line 002. Correspondingly, as shown in Figure 4.6(a), the method *onCreate()* of the MainActivity's super class (*android/app/Activity*) is invoked at Line 2. Note that other unpackers (e.g., [193, 182]) cannot profile such behaviours.

4.4.6 Implementation on ART

We adopt similar methods to monitor packed apps running in ART. Different from DVM that provides only two functions to invoke a Java method in native code, ART provides *CallTYPEMethod()* functions and *CallStaticTYPEMethod()* functions to call non-static methods and static methods, respectively. *TYPE* indicates the type of the method's return value.

To call a native method through JNI, ART invokes the functions *artInterpreterToCom-*

Table 4.2: Wrappers for tracking Dex and ART related events.

Category	Wrapped Functions	Tracked Information
Dex Data	<i>DexFile.DexFile()</i>	Dex file parsing
	<i>ClassLinker.DefineClass()</i>	Class loading
	<i>ArtMethod.LoadMethod()</i>	Method resolution
	<i>ArtMethod.Invoke()</i>	Method execution
	<i>JniMethodStart()</i>	JNI invocation
	<i>JniMethodStartSynchronized()</i>	
	<i>InvokeMethod()</i>	Java reflection
	<i>InvokeWithVarArgs()</i>	JNI reflection
	<i>InvokeWithJValues()</i>	
	<i>InvokevirtualOrInterfaceWithJValues()</i>	
	<i>InvokeVirtualOrInterfaceWithVarArgs()</i>	
Native Module	<i>LoadNativeLibrary()</i>	Native code loading

piledCodeBridge() or *art_quick_generic_jni_trampoline()* depending on whether the native method contains compiled code or not. Both of them eventually establish the JNI calling environment according to the JNI call convention in ART. Before the execution of native code, the functions *JniMethodStart()* or *JniMethodStartSynchronized()* will be called depending on whether the native method is synchronized or not. We wrap these two functions to track the invocation of JNI methods and get the name of JNI methods in ART.

Since ART is implemented in C++, all Dex data structures are stored in class objects. We parse such class objects and recover Dex data structures from them. *PackerGrind* currently supports the ART in Android 6.0 and it reuses the modules at the system layer and instruction layer for DVM.

4.5 Evaluation

We conduct extensive experiments to evaluate *PackerGrind* by answering the following five questions.

Q4.1: *Can PackerGrind be adaptive to the evolution of packers and identify their protection mechanisms?*

Q4.2: *Can PackerGrind correctly recover Dex files?*

Q4.3: *Is PackerGrind better than other available unpackers?*

Q4.4: *Can PackerGrind facilitate the analysis of malware?*

Q4.5: *What is the overhead of PackerGrind?*

4.5.1 Data Set

We use two sets of packed apps to evaluate *PackerGrind*. The first set has 480 packed apps with ground truth. More precisely, we downloaded 40 randomly selected open-source apps from F-Droid [23] and then uploaded them to 6 online commercial packing services (Qihoo [139], Ali [41], Bangcle [56], Tencent [164], Baidu [55], Ijiami [100]) in Mar. 2015 (denoted as DB-15) and Mar. 2016 (denoted as DB-16) to construct 480 packed apps. The second set consists of 200 packed malware samples from Palo Alto Networks[129]. These samples are packed by eleven packers including Ali [41], APKProtect [3], Baidu [55], Bangcle [56], Ijiami [100], Naga [124], Qihoo [139], Tencent [164], LIAPP [156], Netqin [127], and Payegis [132].

We conduct the experiments in both Android 4.4 with DVM and Android 6.0 with ART on a Nexus 5 smartphone [11]. *PackerGrind* monitors the protection patterns of these

Table 4.3: Protection mechanisms adopted by six packers in DB-15 and DB-16. The symbol before (or after) “—” denotes whether a packer in DB-15 (or DB-16) uses the mechanism or not.

Packer	Qihoo	Ali	Bangcle	Tencent	Baidu	Ijiami
Dynamically Release Dex	✓—✓	✓—✓	✓—✓	×—✓	✓—✓	✓—✓
Dynamically Modify Dex	×—✓	×—✓	×—✓	×—✓	✓—×	✓—✓
Customized Dex Parsing	×—✓	×—×	×—×	×—×	×—×	×—✓
Re-implement Method	×—×	×—×	×—×	×—×	×—✓	×—×
Anti-Debug (e.g., <i>ptrace</i>)	×—×	×—×	✓—✓	×—×	×—×	×—✓

packed apps and recovers their Dex files.

4.5.2 Protection Mechanisms

Using *PackerGrind*, we reveal the protection mechanisms adopted by 6 packers, each of which has two versions for DB-15 and DB-16, individually. As shown in Table 4.3, packers are evolving with new techniques and hence unpackers should be adaptive to the evolution. *PackerGrind* can unpack apps protected by all mechanisms except the *Re-implement Method*.

All but Tencent packer of DB-15 release Dex files to memory dynamically. In DB-16, all packers except Baidu dynamically modify selected structures in the Dex file. For example, Ali packer changes the `class_data_item` of each class in the loaded Dex file from invalid value to valid one before the class is defined. Moreover, Ijiami packer sets the Dex file header with valid value before `dexFileParse()` is called, and changes it to invalid value after using `dexFileParse()`.

Table 4.4: Number of runs required for determining all Dex data collection points.

Packer	Qihoo	Ali	Bangle	Tencent	Baidu (DB-15)	Baidu (DB-16)	Ijiami
Number of runs	2	1	1	1	2	1	2

Qihoo packer and Ijiami packer of DB-16 use their own functions rather than the standard runtime functions to parse certain structures of the Dex file. Qihoo packer invokes the native code in its library `libjiagu.so` instead of `dvmDefineClass()` to load classes. Ijiami packer parses the methods of the loaded classes again using the native code in its library `libexec.so`, and changes the instruction offsets of those methods to valid values right before `dvmDefineClass()` returns. Baidu packer of DB-16 re-implements all `onCreate()` functions using native code with the same functionality. Bangle packers uses `ptrace()` to protect the app process from being attached by debugging tools while Ijiami packer of DB-16 periodically searches for the string “@com.android.reverse-” to detect ZjDroid [32].

Answer to **Q4.1**: *PackerGrind* is adaptive to the evolution of packers and can identify the protection mechanisms adopted by various packers.

4.5.3 Recovering Dex Files

Number of runs required for determining all Dex data collection points Table 4.4 shows that *PackerGrind* needs one run for Ali, Bangle, Tencent, and the new version of Baidu packers (i.e., DB-16). It takes two runs to handle Qihoo, Ijiami, and the old version of Baidu packers(i.e., DB-15). Once the Dex data collection points for a packer are identified, *PackerGrind* can recover the Dex files in one run.

Qihoo. Since Qihoo packer parses the protected Dex file using native code in its library `libjiagu.so` instead of standard functions, *PackerGrind* locates the first Dex file

when *dvmIntepret()* is invoked (i.e., the fourth Dex data collection point). From the tracking report, we notice that the `class_data_off` is changed to zero after its library `libjiagu.so` is loaded. Since *PackerGrind* does not find the Dex file at other collection points, we add a new data collection point right before *dvmLoadNative()* for the second run, and then the correct Dex file is recovered.

```

22 public void onCreate(Bundle bundle) {
23     String str = "LXXX;->onCreate001(Landroid/os/Bundle;)V";
24     A.d(str);
25     onCreate001(bundle);
26     A.e(str);
27 }
28
29 private void onCreate001(Bundle savedInstanceState) {
30     super.onCreate(savedInstanceState);
31     setContentView(C0000R.layout.main);
32     this.display = ((WindowManager) getSystemService("window")).getDefaultDisplay();
33     this.mLibrary = GestureLibraries.fromRawResource(this, C0000R.raw.gestures);
34     if (!this.mLibrary.load()) {
35         finish();
36     }
37     findViewById(C0000R.id.gestures).addOnGesturePerformedListener(this);
38 }

```

Figure 4.7: Content of *onCreate001()* after the 2nd run.

Baidu. For the samples packed by the old version of Baidu packer (i.e., in DB-2015), we find that the method *onCreate001()*, which is recovered after the first run, is empty. Hence, we add a new data collection point after *A.d()* by analyzing the tracking report and the Dex file recovered in the first run. In the second run, the Dex file is successfully recovered (e.g., Figure 4.7).

Ijiami. After the first run, we observe that all instructions of the methods in the MainActivity are zero. By analyzing the tracking report, we find that the packed apps modify the instructions of *Methods* after method resolution. Moreover, the instructions of *Methods* are different from those of the corresponding `code_item` structure in Dex file. Therefore, we add a new data collection point after *dvmDefineClass()* for the second run, and then the Dex file is successfully recovered.

Table 4.5: Difference between the original Dex file and recovered Dex file from the samples of DB-15/DB-16 (\oplus , \ominus and \odot represent the recovered Dex file has additional code, less code, and the same code compared with the original Dex file, respectively).

Packer	Qihoo	Ali	Bangcle	Tencent	Baidu	Ijiami
DB-15	\odot	\oplus	\oplus	\oplus	\oplus	\odot
DB-16	\odot	\odot	\oplus	\odot	$\oplus\ominus$	\oplus

Correctness of recovered Dex files We assess the correctness of recovered Dex files from three aspects. First, we apply five popular static analysis tools, which can reverse-engineer Dex files, to the recovered Dex files, because they adopt different verification strategies to check Dex files. These tools include *Baksmali* [24], *Dexdump* [28], *Dex2jar* [27], *Jadx* [30] and *IDA Pro* [29]. *PackerGrind* successfully recovers the Dex files of almost all samples. The only exception comes from *Dex2Jar* when it handles the recovered Dex files from Tencent samples (from DB-15). It failed to transform the Dalvik bytecodes into java bytecodes due to Dex optimization conducted by DVM.

Second, we compare the difference between the original Dex files and the recovered Dex files. We randomly select 60 packed samples (10 samples packed by each packer), decompile the Dex files into Java codes, and then *manually* compare the decompiled Java codes from the original Dex files and those from the recovered Dex files. The comparison results are summarized in Table 4.5. The recovered Dex files are the same as the original Dex files for Qihoo packer, Ijiami packer of DB-15, Ali packer of DB-16, and Tencent packer of DB-16.

Ali packer of DB-15 adds two classes to the original Dex files, each of which has one field and three empty methods. It also inserts the invocation of “*Exit.b(Exit.a())*” to the beginning of every Java method. *Exit.a()* just returns *false* and *Exit.b()* is empty. Tencent packer of DB-15 adds two classes to each packed sample while Ijiami packer of DB-16 inserts five classes.

For Bangcle packer, there are six additional classes and twelve additional classes added to the packed samples of DB-15 and DB-16 respectively. In the main activity class, a method named `com.sec_plugin_action_APP_STARTED()` is inserted and invoked at the beginning of the method `smallempthonCreate()`. Bangcle packer of DB-15 creates an intent named `com.secneo.plugin.action.APP_STARTED` and broadcasts it in `com.sec_plugin_action_APP_STARTED()`. Bangcle packer of DB-16 further creates a new monitoring thread in `com.sec_plugin_action_APP_STARTED()`. Baidu packer of DB-15 adds two classes to each packed sample, and re-implements all the `onCreate()` methods using the dynamic code modification technique. Baidu packer of DB-16 inserts one additional class to each packed app but replaces the implementation of `onCreate()` methods with native codes. Therefore, for these methods, the recovered Java code is less than the original Java code.

Answer to **Q4.2**: *PackerGrind* can correctly recover all Dex code that are not removed by packers as well as the additional classes/methods inserted by packers. Even for the methods that are re-implemented in native code, *PackerGrind* can still recover useful semantic information, based on which it is possible to regenerate the Dex code.

4.5.4 Comparison

While two recent tools, DexHunter and AppSpear, claimed to be general unpackers, only DexHunter's source code is available. Hence, we only compare *PackerGrind*, DexHunter and Android-unpacker[2] using 30 randomly selected samples from six packers (i.e., 5 samples from each packer). We perform two-step checking on the correctness of recovered Dex files. First, the Dex files can be disassembled by Baksmali. Second, we compare them with the Dex files of the original apps. The failure in any step will lead to \times in Table 4.6 indicating unsuccessful unpacking.

Table 4.6: Comparison among Android-unpacker, DexHunter and *PackerGrind*.

Packer	Qihoo	Ali	Bangle	Tencent	Baidu	Ijiami
Android-unpacker [2]	×	×	×	×	×	×
DexHunter [193]	×	✓	✓	✓	×	×
<i>PackerGrind</i>	✓	✓	✓	✓	✓*	✓

Android-unpacker recovers Dex files by attaching to the app process through *ptrace* and dumping Dex files in memory. However, it cannot be attached to packed apps with anti-debugging capability. Moreover, for packers that dynamically release and modify Dex files, Android-unpacker cannot obtain the valid Dex files because it does not know the proper dumping moment.

PackerGrind can successfully recover all Dex files from six packers. We mark Baidu samples with * because the Dex code in *onCreate()* of those samples has been re-implemented in native code as detailed in Chapter 4.4.5. Although *PackerGrind* is not designed to reverse-engineer native code, it can still provide very useful information about the native method as explained in Chapter 4.4.5.

DexHunter cannot correctly recover the Dex files for samples from Qihoo, Baidu and Ijiami. For Qihoo samples, the Dex files dumped by DexHunter only contain stub classes instead of real code, such as *com.qihoo.util.Configuration* and *com.qihoo.util.StubApplication*, because Qihoo packer uses its own functions instead of runtime methods monitored by DexHunter to load classes. For Baidu samples, the Dex files dumped by DexHunter cannot be disassembled because their Dex headers have been modified by the packed apps. Hence, they cannot be recognized by de-compilers. DexHunter also cannot recover the original Dalvik bytecodes of *onCreate()*. For Ijiami samples, DexHunter cannot unpack them successfully due to the time-out checking mechanism utilized by Ijiami. More precisely, the packed apps will check the existence

of a long-running task, which exceeds a time threshold, and exit if found. Therefore, the process of DexHunter will stop because its unpacking operations takes such a long time that the packed app exits quickly.

Answer to **Q4.3**: *PackerGrind* outperforms other available unpacking tools (i.e., Android-unpacker and DexHunter).

4.5.5 Unpacking Malware

We apply *PackerGrind* to 200 malware samples packed by eleven popular packers and successfully recover all Dex files. By performing static analysis on these Dex files, we find that malware often employed packers to hide the invocations of sensitive APIs requiring permissions. Given a Dex file, we scan all sensitive APIs in it, and count how many permissions are required according to the mapping between permissions and APIs from PScout [50]. It is worth noting that many detection systems leverage sensitive APIs and permissions to discover mobile malware [190, 51, 186, 175, 162].

Let P_p and P_r denote the number of permissions required by a packed app and its recovered Dex file, respectively. We calculate the means of P_p and P_r for all malware samples packed by a packer. The result listed in Table 4.7 shows that from the recovered Dex files we find more evidences (i.e., sensitive APIs requiring certain permissions) to explain why a malware sample needs certain permissions. For example, for Naga samples, before unpacking, we cannot find any API invocation requiring the permissions in the manifest (i.e., $P_p = 0$). Malware detection system may think that these samples just overclaim the permissions without using them. In contrast, after unpacking, we can identify API invocations that require 3 permissions in the manifest on average (i.e., $P_r = 3$). Let A_p and A_r indicate the number of sensitive API invocations in a packed app and its recovered Dex file, respectively. We compute the means of A_p and A_r for all

Table 4.7: Permissions and sensitive API calls in malware samples before and after unpacking by *PackerGrind*.

Packer	Ali	Apkprotect	Baidu	Bangle	Ijiami	Naga	Qihoo	Tencent	LIAPP	Netqin	Payegis
Number of samples	26	24	19	34	24	12	27	9	2	18	5
Average value of P_p	0.00	2.65	0.21	0.24	0.00	0.00	2.07	7.22	0.00	4.28	1.80
Average value of P_r	5.38	3.65	10.11	7.56	7.04	3.00	8.81	7.78	0.50	10.33	2.00
Average value of A_p	0.00	5.62	0.95	0.88	0.00	0.00	6.30	40.67	0.00	19.56	5.80
Average value of A_r	49.92	9.38	111.68	50.18	45.88	14.33	88.52	58.00	4.50	90.94	11.00

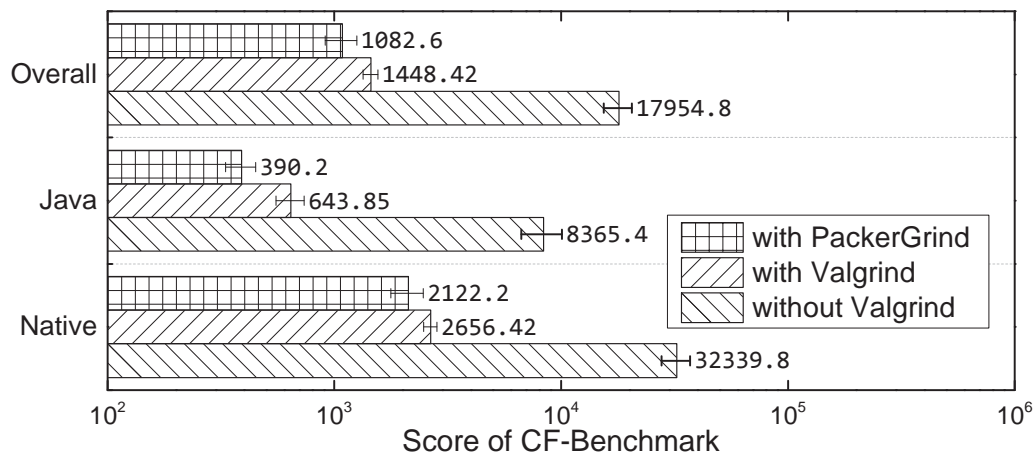


Figure 4.8: CF-Benchmark results.

malware samples packed by a packer. The result listed in Table 4.7 obviously indicates that many more sensitive APIs can be found from the recovered Dex file. For example, no sensitive API invocation is found in malware samples packed by Ijiami whereas on average 45.88 sensitive API invocations can be found from the recovered Dex files.

Answer to **Q4.4**: *PackerGrind* can facilitate malware detection by exposing the hidden malicious components.

4.5.6 Overhead

To evaluate the overhead introduced by *PackerGrind*, we run CF-Benchmark [25] 30 times on Nexus 5 without Valgrind, with Valgrind, and with *PackerGrind*, respectively. The scores of CF-Benchmark on the same smartphone without Valgrind serve as the baseline for comparison. Figure 4.8 shows the results obtained in three scenarios, which include the overall scores, the scores of Java operations, the scores of native operations. We can see that Valgrind incurs 12.4 times slowdown and *PackerGrind* brings 17.6 times slowdown on average compared with the baseline. Since *PackerGrind* is based on Valgrind, it is still efficient because it is only 1.34 times slower than Valgrind. Compared with the dynamic

analysis systems based on emulator that may introduce 11-34 times slowdown[180], *PackerGrind* has acceptable efficiency.

Answer to **Q4.5:** *PackerGrind* introduces acceptable low overhead compared with that of Valgrind and emulator-based dynamic systems.

Chapter 5

Scrutinizing Smartphone-Based Mobile Network Measurement Apps

5.1 Overview

Many apps have been developed to measure the performance of mobile networks. Unfortunately, their measurement results may not be what users have expected, because the results could be biased by various factors and the apps' descriptions may confuse users. Although a few recent studies pointed out several factors, they missed other important factors and lacked fine-grained analysis on the factors and measurement apps. Moreover, none has studied whether or not the descriptions of such apps will mislead users. In this thesis, we conduct the *first* systematic study of the factors that could bias the result from measurement apps and their descriptions. We identify new factors, revisit known factors, and propose a novel approach with new tools to discover these factors in proprietary apps. We also develop a new measurement app named MobiScope for demonstrating how to mitigate the negative effects of these factors. Furthermore, we construct enhanced descriptions for measurement apps to provide users more information about what is measured. The extensive experimental results illustrate the negative effects

of various factors, the improvement in performance measurement brought by MobiScope, and the clarity of the enhanced descriptions.

With respect to performance measurement, our major contributions in this chapter are summarized as follows:

- We conduct the *first* systematic study of the factors that could bias the result from measurement apps by identifying new factors, revisiting known factors, and quantifying the negative effects through extensive experiments.
- We propose a novel approach combining static bytecode analysis and dynamic trace analysis, and develop practical tools to facilitate discovering these factors in apps.
- We develop MobiScope, a new measurement app that adopts various techniques to mitigate those factors' negative effects.
- We perform the *first* examination on the clarity of measurement apps' descriptions, and construct enhanced descriptions to inform users what is measured.

The rest of this chapter is organized as follows. Chapter 5.3 elaborates the classes of factors. Chapter 5.4 details `AppDissector` and `AppTracer` for inspecting the factors. Chapter 5.5 describes MobiScope. We present experiments in Section 5.6, and report user studies in Chapter 5.7.

5.2 Background

5.2.1 Mobile Network Measurement

Recent studies demonstrated that the measurement result from the mobile measurement apps may be biased [177, 114]. For example, the round-trip time (RTT) measured by an app is not exactly the network RTT. According to RFC 2681 [42], the RTT reported

Listing 5.1: RTT measurement code

```
1  /* Run HTTP ping measurement task */
2  public long Measurement() {
3      HttpRequestBase request;
4      ... // Initialize request
5      long startTime = System.currentTimeMillis();
6      HttpResponse response = httpClient.execute(request);
7      StatusLine statusLine = response.getStatusLine();
8      ... // Parse HTTP response
9      long duration = System.currentTimeMillis() - startTime;
10     return duration; // Return measurement result
11 }
```

by an app is determined by the *host times*, which include the timestamp just prior to sending the packet and that right after receiving the response packet, whereas the network RTT is calculated using the *wire times*, which refer to the time when the packet leaves the smartphone's network interface (NIC) and the time when the corresponding response packet arrives at the smartphone's NIC. Our previous work showed that the difference between *host times* and *wire times* is not ignorable because of the delay noise introduced by the device, such as context switch in operating system (OS), Dalvik virtual machine (DVM) in Android system, etc. [177]. Recent study from Li et al. [114] validated our observation and reported that the delay due to DVM and Linux kernel is around 2.4ms. As another example, when measuring network capacity, the probing packets are expected to be sent out back-to-back (i.e., zero spacing between probing packets) [188, 122]. Unfortunately, we observed that the delay noise from the device will notably enlarge the spacing[177].

5.2.2 A Motivating Example

We use a real example to demonstrate how to determine whether an app is affected by certain pitfall (e.g., P2 of HTTP ping in Section 5.3.1) by performing both static bytecode analysis and dynamic trace analysis.

The code of the HTTP ping function is shown in Listing 5.1, where the Java method

execute() in the class `Android.net.http.AndroidHttpClient` is invoked to send an HTTP request and receive the corresponding response (i.e., line 6). Two timestamps are recorded before and after the invocation of *execute()* (i.e., lines 5 and 9) and they are used to calculate the RTT at Line 9. The two timestamps are obtained through invoking the method `System.currentTimeMillis()` in Line 5 and Line 9, respectively. Besides, the duration (i.e., RTT result) is also calculated in Line 9, and the RTT result returns in Line 10.

However, the Java method *execute()* actually involves several steps, including performing DNS lookup, creating socket, establishing a TCP connection, setting socket options, sending the HTTP request, and receiving the corresponding HTTP response. Since establishing the TCP connection requires one RTT, it will cost at least 2 RTTs for finishing the method *execute()*. In other words, the code in Listing 5.1 is affected by the pitfall and will lead to wrong RTT values.

5.3 Factors Affecting Measurement Results

We classify the factors that could bias the measurement into three categories, namely implementation patterns (Section 5.3.1), Android architecture and configurations (Section 5.3.2), and network protocols (Section 5.3.3). We identify new factors in categories 1 and 2, including implementation patterns for HTTP/TCP based measurement, monitoring and power management mechanisms in Android. Moreover, we revisit known factors, including multiple-layer nature of Android in category 2 and mobile and WiFi protocols in category 3. More precisely, in contrast to previous study [114], we conduct a fine-grained analysis on the effect of each layer and examine the new Android runtime. For category 3, we examine the protocols' effect on the measurement result and suggest solutions.

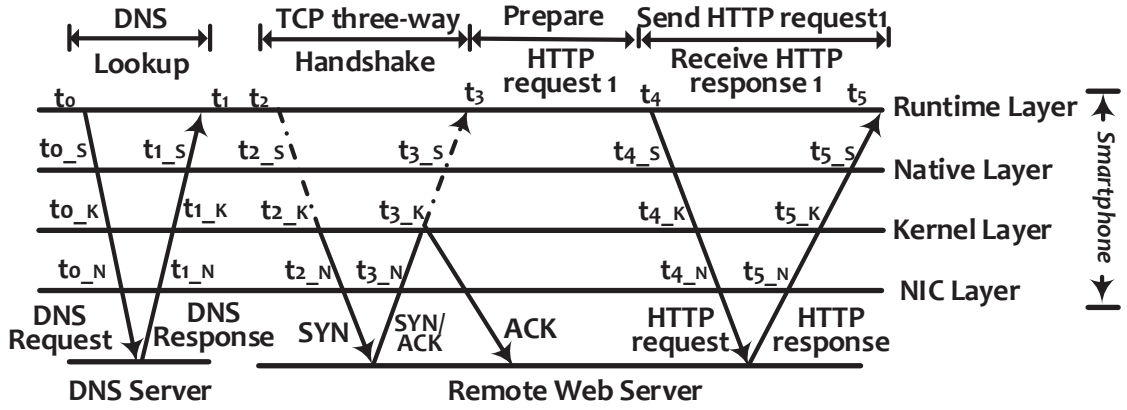


Figure 5.1: An example of RTT measurement conducted on an Android smartphone. We use dash-dot line to connect t_2 and t_{2-k} because the TCP SYN packet is sent by the kernel but triggered by the function at the runtime layer. Similarly, the dash-dot line connecting t_{3-k} and t_3 indicates that the TCP SYN/ACK packet is received by the kernel without forwarding to the runtime layer but the system will notify the function at that layer.

We use the process of RTT measurement, one primitive measurement task[42], to explain the effect of various factors. These factors have similar effect on the capacity and throughput measurement, because they will introduce non-negligible time gap to back-to-back packets used for measuring these metrics[122]. Note that although the effect of some factors could be mitigated by using statistical algorithms (e.g., outlier detection) [128], eliminating the effect of other factors requires app re-design or modification (e.g., those in Section 5.3.1).

Figure 5.1 shows the measurement process with three stages, namely, (1) DNS lookup; (2) TCP three-way handshaking; and (3) preparing and sending HTTP request 1 to a remote web server, and receiving HTTP response 1. Figure 5.1 also depicts the multiple layers of Android, including runtime, system (i.e., the user space of Android’s customized Linux), kernel, and network interface (NIC). In Figure 5.1, t_i denotes the timestamp obtained at the runtime layer. t_{i_S} , t_{i_K} , and t_{i_N} ($i=0 \dots 9$) represent the timestamps recorded at system/kernel/NIC layer, respectively. According to RFC2681, t_i , t_{i_S} , and t_{i_K} are *host times*, referring to the timestamps acquired right before sending a request and those obtained just after receiving the response at different layers. t_{i_N} is the wire time,

including the time when a packet leaves the NIC and the time when the response packet arrives at the NIC.

5.3.1 Category 1: Implementation Patterns

Although various apps claim measuring the same metric (e.g., RTT), they may adopt different implementation patterns that lead to different results. We summarize common patterns used by apps for measuring RTT in Figure 5.2. Generally, an app first obtains a timestamp (i.e., t_{Start}), and then performs the measurement. Finally, it records another timestamps (i.e., t_{End}) and computes RTT as $t_{End} - t_{Start}$. Note that we only examine HTTP (or TCP) based RTT measurement, because most measurement apps support them and it is easy for users to find a web server for measurement, not to mention that HTTP is widely used by various apps[184].

HTTP-based RTT measurement: H-P1 measures $t_3 - t_0$ if resolving the destination's IP is required. Otherwise, it outputs $t_3 - t_2$. More precisely, after t_{Start} is gained, an instance of `URLConnection` is created through `openConnection()` in the class `java.net.URL`. Then, `connect()` in the class `java.net.HttpURLConnection` is called before getting t_{End} . Note that the Java method `connect()` calls the native method `getaddrinfo()` to perform DNS lookup, and then invokes the native method `connect()` to create a TCP connection.

H-P2 measures $t_5 - t_0$ if IP resolving is needed. Otherwise, it estimates $t_5 - t_2$. Specifically, between getting t_{Start} and t_{End} , it employs `execute()` in the class `org.apache.http.client.HttpClient` to send an HTTP HEAD request and receive the corresponding HTTP response. Note that before sending an HTTP request and receiving the HTTP response, `execute()` calls the native methods `getaddrinfo()` and `connect()` to establish a TCP connection.

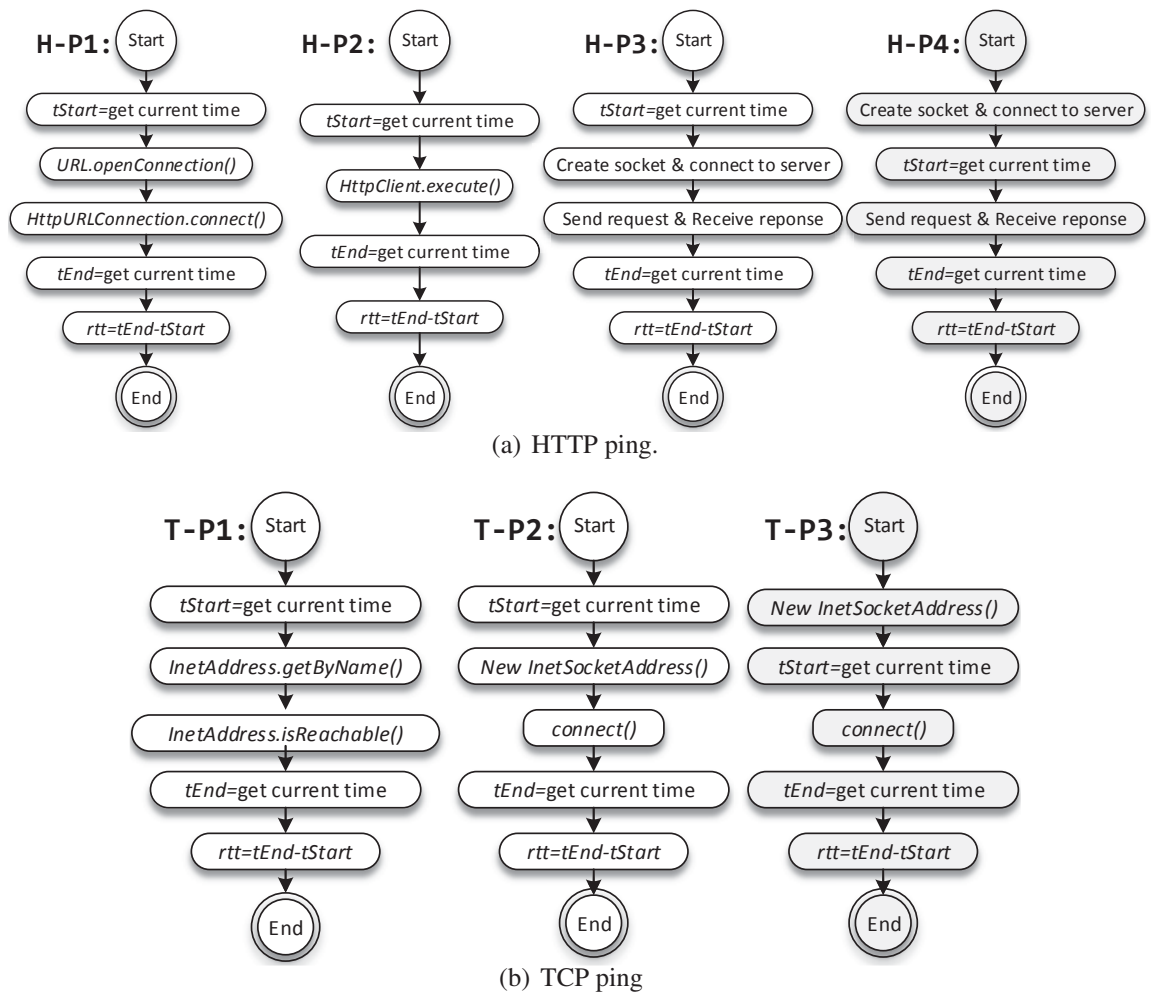


Figure 5.2: Implementation patterns for HTTP-based and TCP-based RTT measurement.

H-P3 measures $t_5 - t_2$ using methods in the class `java.net.Socket`. The whole process includes establishing the TCP connection, constructing an HTTP request, sending the HTTP request, receiving the HTTP response, and parsing the HTTP response.

H-P4 estimates $t_5 - t_4$, which is the correct network RTT measured by HTTP. To minimize the difference between *host time* and *wire time*[42], $tStart$ is captured just before sending the request encapsulated in *one* packet, and $tEnd$ is recorded right after receiving the first packet of the response.

TCP-based RTT measurement: **T-P1** measures $t_3 - t_0$ if IP resolving is required. Otherwise, it yields $t_3 - t_2$ or a much larger value depending on the implementation of *isReachable()*. More precisely, after using *getByName()* in the class `java.net.InetAddress` to resolve the IP, it invokes *isReachable()* in the same class for checking whether the destination is reachable. According to official documentation [1], *isReachable()* first uses ICMP to test the reachability, and returns to TCP (i.e., exploit TCP three-way handshaking for RTT measurement) if ICMP method fails. Therefore, if ICMP packets are dropped by a router, which is common in today's Internet, the measured value would be much larger than the real RTT. Although we find that the current implementation of *isReachable()* does not realize the ICMP-based probing, it is *not* recommended to use this method for measuring RTT in case the ICMP-based probing is realized in future version.

T-P2 invokes two Java methods: *InetSocketAddress()* in the class `java.net.InetSocketAddress` and *connect()* in the class `java.net.Socket`. Since the former method will conduct DNS lookup, **T-P2** measures $t_3 - t_0$ if IP resolving is needed. Otherwise, it outputs $t_3 - t_2$. To accurately measure network RTT, we suggest recording *tStart* and *tEnd* right before and after invoking *connect()* to avoid the additional delay due to DNS lookup, as shown in **T-P3** in Figure 5.2(b).

5.3.2 Category 2: Android Architecture and Configurations

Multiple-layer nature of Android. Since apps run within its runtime, which is a Linux process, packets sent from apps would be delayed at all layers. Since Android 5.0, the default runtime is changed from the Dalvik virtual machine (DVM) to the new Android runtime (ART) for better performance [4]. In particular, apps will be compiled into native code before execution. Note that 51.6% Android devices are still using DVM [31].

Although Li et al. found that DVM may introduce considerable delay [114], they neither conduct fine-grained analysis on the delay caused by different layers nor study ART.

Monitoring mechanisms. Several monitoring mechanisms can be used in Android for monitoring packets. *libpcap* captures packets in the kernel through BPF filter. *Netfilter* allows users to register packet handlers and enables *iptables* to inspect packets that match pre-specified rules. Using *iptables*, *VpnService* [19] allows apps to redirect traffic to a tunnel. It has been employed to capture packets and conduct measurement [146]. All these mechanisms will introduce additional delay if they inspect the measurement packets.

Power management. Android has an aggressive power management strategy but provides a mechanism called *wake locks* [10] that empowers apps to keep the device awake. These mechanisms have an indirect impact on measurement results because they affect the parameters of PSM (Power Save Mode) adopted by WiFi interface, which invites additional delay.

5.3.3 Category 3: Network Protocols

The effect of network protocols on measurement is mainly due to their state transitions, because previous studies reveal that the state transitions will lead to noticeable delays [90, 148, 135]. We revisit these factors because to what extent it may bias the measurement result remains unknown.

Cellular Network. The RRC (Radio Resource Control) protocol of cellular networks influences power consumption and network performance [90, 148]. Typically, 3G has three main RRC states (i.e., IDLE, FACH and DCH), while LTE has two main states (IDLE and CONNECTED). Typically, packets transmitted by cellular interface in the DCH (3G) or CONNECTED (LTE) state experience the shortest delay. Therefore, when the cellular

interface is in the states other than `DCH` or `CONNECTED`, RTT measurement will suffer from additional delays due to the state transition.

WiFi Network. PSM allows WiFi interfaces to sleep for an integer number of beacon intervals (i.e., *ListenInterval*), and then wake up to detect the presence of its buffered frames in the access point (AP) [96]. If no frames are detected, the interface continues to sleep for *ListenInterval* beacon intervals. Otherwise, it wakes up to retrieve the buffered frames one by one. When none of the buffered frames are left, the WiFi interface goes back to sleep again. Pyles et al. found that different PSM algorithms lead to different delays [135].

5.4 Is an App Affected by These Factors?

We combine static bytecode analysis and dynamic trace analysis to inspect whether an app uses any implementation pattern in Figure 5.2, and whether it adopts any approaches to mitigate the negative effect of the factors described in Chapter 5.3. We further develop two tools (i.e., `AppDissector` and `AppTracer`) to facilitate this inspection. Given a measurement app, `AppDissector` locates its measurement code and checks: **(C1)** whether it uses native code to perform the measurement for avoiding the effect of runtime; **(C2)** whether it employs *VpnService* to handle packets; **(C3)** whether it requests *wake lock* and *Wi-Fi lock* to mitigate the effect of power saving mechanisms. `AppTracer` collects information of method calls in Android framework, system libraries, and system calls to construct the cross-layer method call graph and determine whether the app uses *libpcap* or *iptables*.

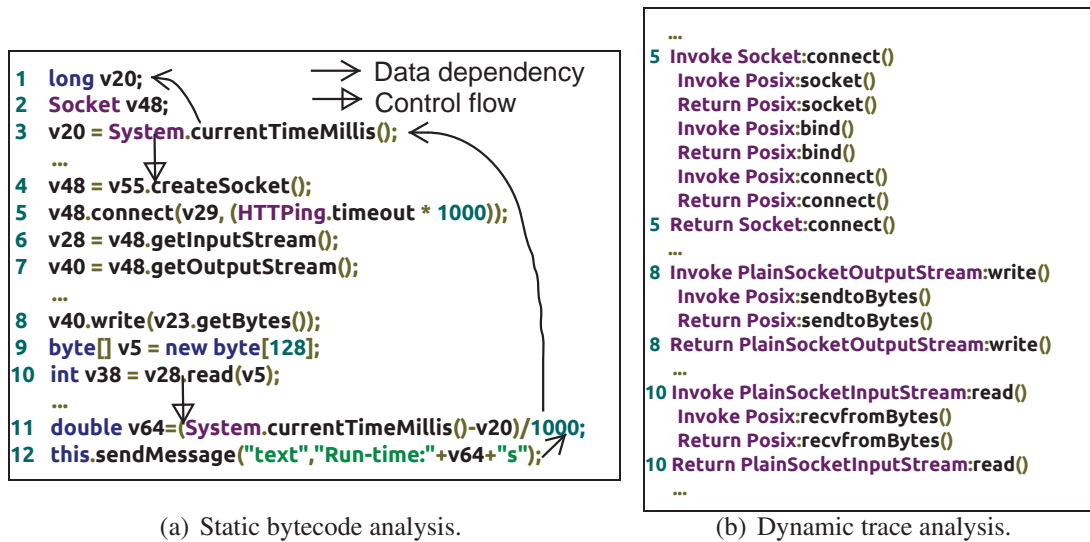


Figure 5.3: A snippet of the static and dynamic analysis results of HTTPing.

5.4.1 Static Bytecode Analysis

Pre-processing. Given an app, AppDissector uses VulHunter[136] to its abstract syntax trees(AST), inter-procedure control flow graph(ICFG), method call graph, and system dependency graph, and utilizes ICCTA [113] to find the target of each intent because an app’s components can communicate through intents.

Then, we look for the entry of the measurement process. If the entry is a UI component, we locate its callback functions from two sources, because they may start the measurement process. First, we parse the layout file to find the callback functions because they can be set in it. Second, since an app can get a UI component in code and register event listeners, we first obtain the UI component’s ID and then enumerate all statements using this ID as parameters. After that, we search the AST of each event listener to locate the callback functions.

Locating measurement code. The measurement code includes the measurement result related code for deriving the result and the measurement procedure related

code for conducting the measurement. We first look for the former by locating the variables/fields representing the measurement results and then performing *backward slicing* and *chopping* [110]. More precisely, we conduct *backward slicing* to identify the statements that influence the computation of measurement results. We perform the depth-first traversal from the statement that gets the measurement results following the data dependency relation. The traversal stops at local declaration statement because it defines a new variable without depending on other statements. All statements on the paths are saved in W . Then, we perform *chopping* to identify the statements that use the variables defined in W . For each statement in W , we check the variable defined in it. If the variable is used by another statement s , we add s into W . Finally, all statements in W are regarded as the measurement result related code.

After that, we leverage the statements in W to look for the measurement procedure related code. More precisely, we locate the first and the last statements without considering local declaration statements, and then traverse the control flow graph to find the paths between them. All statements on the paths are saved in W' . We output the statements in W and W' , and regard the statements that are not included in W as the measurement procedure related code.

We use `HTTPing`, whose code snippet is shown in Figure 5.3(a), as an example to illustrate the procedure. In `HTTPing`, the measurement result $v64$ is calculated at line 11. To know how this result is generated, we add line 11 into W and perform *backward slicing* from it. The depth-first traversal stops at the local declaration statement (i.e., line 1). We also add lines 1 and 3 to W . To know how the measurement result is used, we perform *chopping* on each statement in W . As $v64$ used in line 12 (i.e., show user the measurement result), we add line 12 into W . Finally, W contains lines 1, 3, 11, 12. We regard them as the measurement result related code. Then, we traverse from line 3. The traversal stops at line 12. Lines 3-12 are put in W' . Since lines 4-10 are not included in W , we regard them

as the measurement procedure related code.

C1: We traverse the ICFG from the entry of the measurement process to identify statements invoking native methods that affect the measurement results. If found, we use dynamic trace analysis to confirm whether the native methods perform the measurement or not, because our static bytecode analysis module cannot handle native code currently.

C2: We check whether the app requests the permission `BIND_VPN_SERVICE` in `AndroidManifest.xml` and invokes `android.net.VpnService.establish()` to create a VPN interface. If so, the app uses `VpnService`.

C3: An app can keep the screen on by calling `Window.addFlags()` with `FLAG_KEEP_SCREEN_ON` or setting the attribute `android:keepScreenOn` to “true” in the layout file. We look for them by inspecting the ASTs and the layout file. To lock the Wi-Fi, the app must request `WAKE_LOCK` permission and invoke `WifiLock.acquire()`. We discover it by parsing the manifest file and checking the ASTs.

5.4.2 AppTracer: Dynamic Trace Analysis

It is non-trivial to design a tool running in smartphone to collect information about method calls across layers. We accomplish it by developing `AppTracer` based on `Valgrind` [126], whose architecture is shown in Figure 5.4. The tracers trace method invocations and returns at the corresponding layers, and the trace analyzer generates the control flow information based on the logs generated by the tracers.

DVM Runtime Tracer. The information of each invoked DVM function can be collected from `dvmMethodTraceAdd()` if Android’s profiling framework is enabled [179]. Therefore, the DVM runtime tracer wraps this function for tracing functions at the DVM layer. When a measurement app is launched, we enable Android’s profiling framework, and

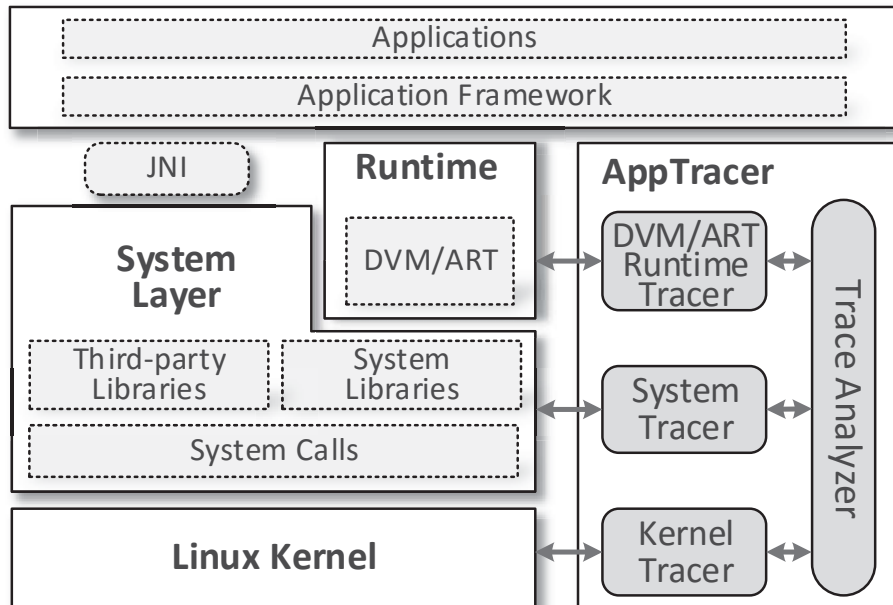


Figure 5.4: Architecture of AppTracer

then the DVM runtime tracer collects the information of method invocation and return in the wrapper function of *dvmMethodTraceAdd()*.

ART Runtime Tracer. AppTracer supports ART. Since the functions *Trace::MethodEntered()* and *Trace::MethodExited()* are called when each method is invoked and returned, respectively, the ART runtime tracer obtains the entering and exiting events of the involved methods by wrapping both functions in `libart.so` and enabling Android's profiling framework.

System Tracer. It collects information about system library functions and system calls. By using `Valgrind` to get the instruction level information and the addresses of all functions in loaded libraries, it obtains the function invocation and return information by matching function addresses and the target addresses of jump instructions. Since `Valgrind` can obtain the number of each system call when it is invoked and returns, we maintain the relationship between the number and the name of each system call to identify all system calls involved.

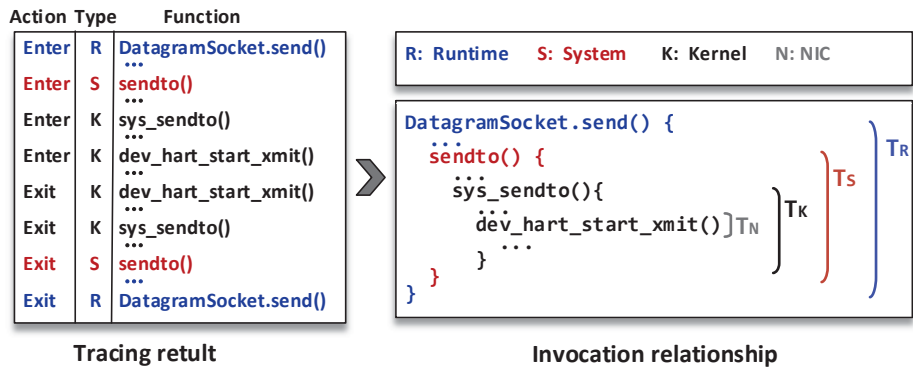


Figure 5.5: Example of reconstructing invocation relationship from tracing result.

Kernel Tracer. It utilizes the *function_graph* tracer of *ftrace* to record the flow of every function call in the kernel and then constructs the function call graph according to the traces generated by the *function_graph* tracer.

Trace Analyzer. It constructs the cross-layer control flow by exploiting the order of entering and exiting a function, which are recorded by tracers at different layers. For example, Figure 5.5 shows the tracing results generated by AppTracer. Since these functions are not executed in parallel, we reconstruct the cross-layer dynamic control flow as shown in the sub-figure on the right side of Figure 5.5 according to the entering and exiting orders of all invoked functions at different layers.

With the cross-layer control flow, we further correlate it to the result of static analysis. For example, from the dynamic analysis results of HTTPing shown in Figure 5.3(b), we match Java methods in Figure 5.3(a) to JNI functions in `libcore.io.Posix.cpp`. Therefore, by combining the results of static and dynamic analysis, we learn that the RTT measured by HTTPing includes constructing TCP connection, sending HTTP request, receiving HTTP response. In other words, it follows the H-P3 pattern in Figure 5.2.

To check whether an app uses *iptables*, AppTracer calls “*iptables -L*” to list all *Netfilter* rules. Since *libpcap* captures packets through `PF_PACKET` socket, AppTracer

hooks the function *socket()* to monitor whether `PF_PACKET` socket is created. If so, `AppTracer` checks whether the socket is used for receiving packets. Based on the result, `AppTracer` knows whether the app uses *libpcap* to capture packets.

5.5 MobiScope

We develop `MobiScope` to demonstrate how to mitigate the effects of the factors in Chapter 5.3. To mitigate the effects of multiple layers (i.e., shorten the difference between *host time* and *wire time* [42]), `MobiScope`'s two major modules (i.e., `kping` and `kband`) are realized in kernel. Although installing kernel modules requires root privilege, we develop them for users requiring highly accurate measurement results.

Using ICMP packets to measure RTT, `kping` first constructs an echo request packet in kernel and stores it in the structure *sk_buff*, and then sends it out by calling kernel function *dev_hard_start_xmit()*, which is the entry of the device driver [177]. It uses *ktime_get_ts()* to obtain the sending timestamp with nanosecond resolution. `kping` registers a *Netfilter* function to receive the echo reply packet. Before the measurement, `kping` calls the function *net_enable_timestamp()* to enable *sk_buff* timestamping so that each packet's arriving timestamp will be stored in the field *tstamp* of structure *sk_buff* by the NIC driver. To measure capacity or throughput, `kband` sends packet trains through *dev_hard_start_xmit()* and receives packets through another registered *Netfilter* function. It records the timestamps by calling *ktime_get_ts()*.

Although `MobiScope` uses *Netfilter*, we minimize its impact on the measurement by taking two measures: first invoking functions in the device driver to send packets for excluding the impact of *Netfilter*; second manipulating the *Netfilter* chains to register the packet receiving function at the first position for mitigating the impact of other *Netfilter*

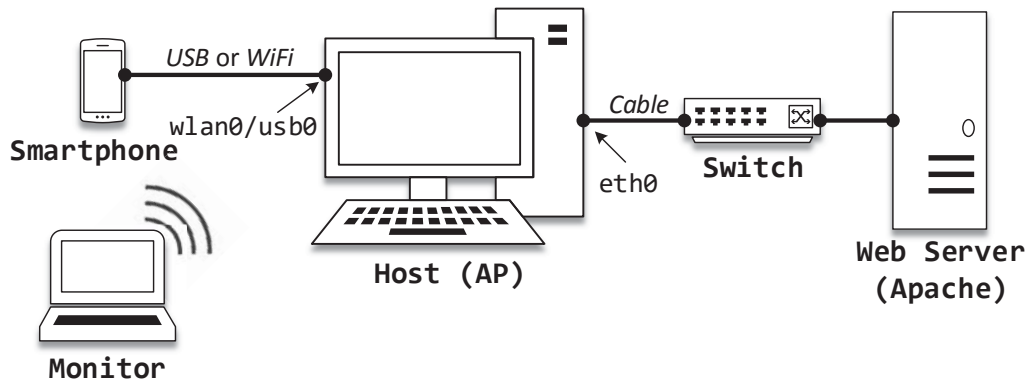


Figure 5.6: The testbed.

rules.

To mitigate the effect of power management and NIC state transition, MobiScope acquires WakeLock and WifiLock to keep the device and WiFi radio awake, respectively. It also sends several packets before starting the measurement to assure that the WiFi state or the RRC state is at the awake state and the `DCH(3G)/CONNECTED(LTE)` state, individually.

MobiScope also includes modules that realize patterns **H-P4** and **T-P3** for conducting HTTP and TCP based RTT measurement at system layer.

5.6 Evaluation

5.6.1 Testbed

As shown in Figure 5.6, the smartphone communicates with an Apache2 web server via a host. The host offers the smartphone two access methods, namely, USB tethering and WiFi (i.e., an AP), and it uses Linux traffic control (TC) and `netem` to emulate various capacity and additional delay. USB tethering offers stable wired connection with very low latency. In contrast, WiFi channel may have variable and large delay due to contention and signal variance. Hence, when examining factors except WiFi state transitions, we connect

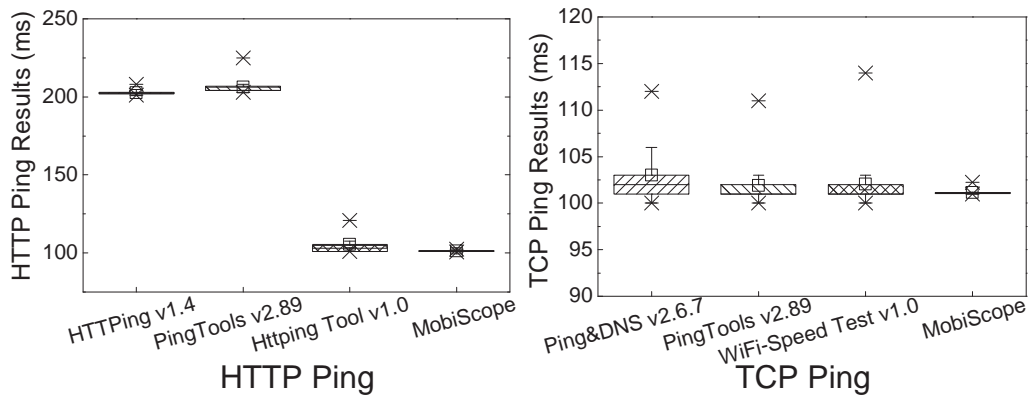


Figure 5.7: HTTP/TCP-based RTT measurement by different ping apps.

the smartphone to the host via USB tethering to avoid unexpected noise. Otherwise, we use WiFi access. Besides collecting timestamps in Android, we also record the timing information of packet transmissions at all possible vantage points in the testbed.

We have two smartphones: Samsung S3 with Exynos 4412 Quad and Murata M2322007 WiFi module, and LG Nexus 5 with Qualcomm MSM8974 Snapdragon 800 and Broadcom BCM4339 WiFi module. The LG Nexus 5 runs either official Android 4.4 or Android 6.0 for DVM or ART. The Samsung S3 runs CM-11.0 (based on Android 4.4) or CM-13.0 (based on Android 6.0), because AOSP builds only support Nexus devices. The apps under investigation including Fing, MobiPerf, Netalyzr, WiFi Speed Test, Internet Speed Test, Ping&DNS, PingTools, he.net, HTTPing, and Httping Tool. These tools cover ten different popular network measurement apps. Their implementation patterns identified by our analysis are summarized in Table 5.1.

5.6.2 Effect of Factors in Category 1

In this experiment, we add 100ms delay to the USB connection between the smartphone and the host. Then we run the measurement applications sequentially and repeat for 50 times. Thus each app returns 50 measurement results for comparison. The left subfigure

Table 5.1: The implementation patterns of 10 measurement apps under examination. ping refers to using the default ping tool in Android.

App	Fing ²	MobiPerf ²	Netlyzt	WiFi Speed Test ¹	Internet Speed Test ¹	Ping&DNS ^{1,2}	PingTools	he.net	HTTTPing	Httping Tool
Implementation	ping	H-P1, T-P1, ping	T-P2	T-P1	T-P2	T-P2, ping	H-P2, T-P2, ping	ping	H-P3	T-P3 ³

¹ WiFi Speed Test and Ping&DNS provide option to prevent screen off; Internet Speed Test keeps screen on during the measurement process.

² Fing, MobiPerf and Ping&DNS acquire the WAKE_LOCK permission in their AndroidManifest.xml file.

³ Its implementation is similar to the pattern T-P3 with difference that it sends HTTP request before getting *tEnd*.

of Figure 5.7 shows the results of HTTP ping from HTTPing, PingToos, Httping Tool, and MobiScope. The right subfigure of Figure 5.7 illustrates the results of TCP ping from Ping&DNS, PingTools, WiFi-Speed Test, and MobiScope. Since all apps use the default ping to conduct ICMP ping, we compare it and MoboScope in Section 5.6.5.

For HTTP ping, the results from HTTPing and PingTools are around *twice* of the real RTT. It is because HTTPing and PingTools use the patterns **H-P2** and **H-P3** respectively. Hence, their results include the time for establishing TCP connection and the time for sending request and receiving response. Although Httping tool claims using HTTP request and response to measure RTT, its implementation is similar to pattern **T-P3** except that it sends HTTP request *before* getting *tEnd*. Therefore, its result is similar to that from MobiScope but has larger variance.

For TCP ping, Ping&DNS and PingTools adopt patten **T-P2** and WiFi Speed Test follows pattern **T-P1**. Their results are similar to that from MobiScope. The reason is we use the web server's IP address instead of domain name so that apps do not need to perform DNS lookup. In other words, they measure $t_3 - t_2$ in Figure 5.1. This also explains why HTTPing and PingTools get the similar results.

Summary Implementation patterns may obviously bias the measurement result. Developers had better describe the implementation patterns of apps to avoid confusing the users.

5.6.3 Effect of Factors in Category 2

Measuring the delays at different layers. To profile the time for delivering a packet across different layers, we construct a cross-layer method call graph using AppTracer,

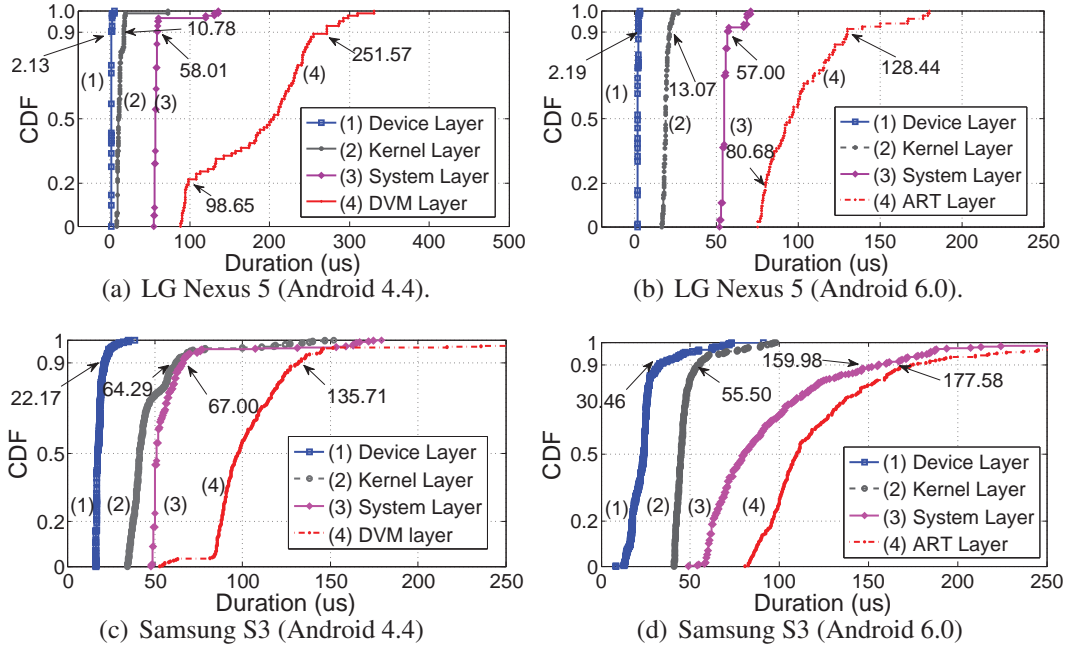


Figure 5.8: Time consumed for sending a UDP packet across different layers.

and then select methods as the entries of different layers. Note that we use the execution time of an entry function to approximate the time used to deliver a packet at the corresponding layer.

To minimize the noise due to timestamp acquiring functions, we get the timestamps used for profiling a Java method from the system layer instead of the runtime layer. More precisely, given a Java method, we obtain the timestamps right before and after its execution by modifying *dvmInterpret()* (in *libdvm.so*) and *ArtMethod.Invoke()* (in *libart.so*) for DVM and ART, respectively. Moreover, we invoke the Java method through Java reflection so that this method will be called and returned in *dvmInterpret()* or *ArtMethod.Invoke()*, individually. Then, at runtime and system layers, we acquire timestamps through *gettimeofday()*. At kernel and device layers, we get them through kernel function *ktime_get_ts()*.

Delay due to Android Architecture. To profile the time consumed for delivering a UDP

packet across different layers, we select *DatagramSocket.send()*, *sendto()*, *sys_sendto()* and *dev_hard_start_xmit()* as the entries of runtime/system/kernel/NIC layer, as shown in Figure 5.5. *sys_sendto()* and *dev_hard_start_xmit()* are kernel functions. The former is the kernel implementation of *sendto()* while the latter delivers the packet to the NIC driver. Moreover, the delays at runtime/system/kernel/NIC layer correspond to $T_R/T_S/T_K/T_N$ in Figure 5.5.

Figure 5.8 shows the CDF of the time consumed at different layers for sending a UDP packet from apps to network. This experiment is repeated for 100 times. We can see that the delays at the device, kernel and system layers are relatively stable and small. For example, Figure 5.8(a) shows that 90% delays are less than 2.13us/10.78us/58.01us at the device/kernel/system layers for Android 4.4. Similarly, Figure 5.8(b) illustrates that 90% delays are less than 2.19us/13.07us/57.00us at the device/kernel/system layers for Android 6.0.

DVM and ART introduce longer delay than other layers, and their values are not stable. Figure 5.8(a) demonstrates that in DVM 20% delays are less than 98.68us and 70% delays are in the range of [98.68, 251.57] us. ART usually causes shorter delay than DVM. Figure 5.8(b) shows that in ART 20% delays are smaller than 80.68us and 70% delays are in the range of [80.68, 128.44] us. The unstable delays caused by the runtime maybe due to the kernel's process scheduling.

Delay due to monitoring mechanisms. We send UDP packets of [628, 1428] bytes from the runtime layer or the system layer, and measure the throughput with/without certain monitoring mechanisms. Table 5.2 shows that *libpcap* and *Netfilter* significantly degrade the throughput due to additional delays.

Netfilter. We insert 10 string matching rules into *iptables* before conducting the experiments. Table 5.2 shows that the *Netfilter* rules result in throughput degradation no

Table 5.2: Packet transmission capacity measured with different configurations (normal/with libpcap/with 10 Netfilter rules).

System		Android 4.4			Android 6.0				
Device		Samsung S3 (Mb/s)		LG Nexus 5 (Mb/s)		Samsung S3 (Mb/s)		LG Nexus 5 (Mb/s)	
Runtime	600 byte	54.99±6.1 / 14.86±5.3 / 12.57±4.4		68.04±5.4 / 44.98±5.3 / 32.87±5.3		37.84±4.1 / 9.00±4.3 / 7.08±3.3		65.03±5.7 / 45.35±4.5 / 26.22±5.1	
layer	1400 byte	87.84±8.3 / 26.56±5.2 / 27.98±3.2		134.45±8.1 / 75.21±9.2 / 70.28±8.3		75.35±9.1 / 11.12±3.2 / 15.51±4.1		136.40±9.2 / 77.41±9.3 / 68.42±7.2	
System	600 byte	90.36±8.9 / 41.40±4.4 / 37.70±5.2		112.46±7.3 / 60.94±7.1 / 32.30±8.9		32.89±8.3 / 15.48±3.3 / 11.77±3.9		72.53±7.3 / 49.42±6.2 / 38.56±7.9	
layer	1400 byte	167.03±9.2 / 56.92±5.3 / 66.91±4.9		294.11±13 / 127.35±9.2 / 73.90±9.3		84.17±7.8 / 34.85±5.1 / 28.50±4.6		215.37±10 / 85.60±4.9 / 75.03±4.6	

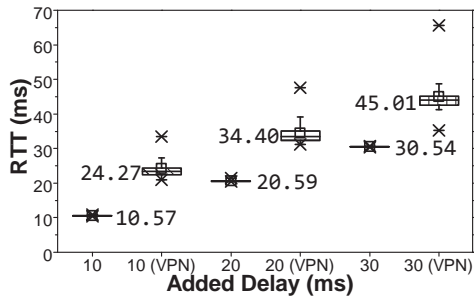


Figure 5.9: RTT measured with and without *VPNservice*.

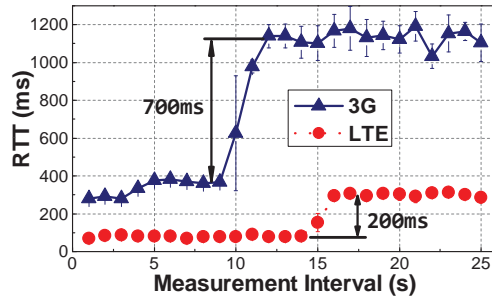


Figure 5.10: RTT measured with different intervals (Cellular).

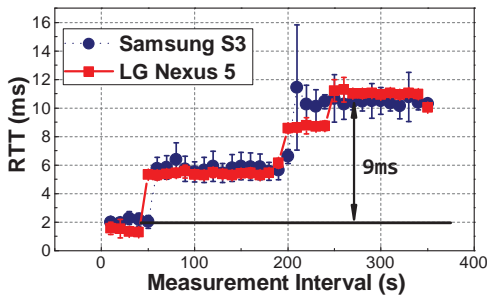


Figure 5.11: RTT measured with different intervals (WiFi).

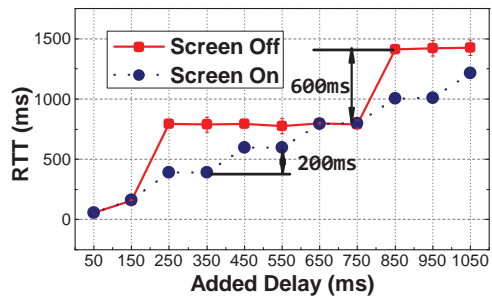


Figure 5.12: RTT measured with different added delays (WiFi).

matter the measurement is conducted at the runtime or the system layer. For example, in LG Nexus 5 running Android 6.0, the throughput can achieve 136.40Mb/s and 215.37Mb/s at the runtime and the system layer, respectively, if the 1428-byte packets are used. However, if the *Netfilter* rules are applied, the throughput drops to 68.42Mb/s and 75.03Mb/s, individually.

Libpcap. Since *Tcpdump* uses *Libpcap* to capture packets, we turn it on or off for getting the results with or without *Tcpdump*. Table 5.2 illustrates that *Tcpdump* also brings obvious overhead to the packet transmission. For example, the throughput measured by 1428-byte packets at the Android 4.4/6.0 runtime layer in LG Nexus 5 is 134.45/136.40Mb/s without *Tcpdump*, but the value drops to 75.21/77.41Mb/s when *Tcpdump* is used. Its effect is less than that of *Netfilter*.

VpnService. We launch *LocalVPN* to measure the effect of *VpnService*. Since preliminary

experiment shows that *VpnService* will significantly delay or drop packets, we only evaluate the additional delay caused by it. More precisely, to emulate different network conditions, we add [10,20,30] ms delay and use MobiScope to measure RTT 100 times. Figure 5.9 shows that *VpnService* causes large additional delay to the results. The differences between the mean values of the measurement results with/without VPN is around 14ms.

Summary The cross-layer nature of Android brings non-negligible delay to the measurement results for both DVM and ART. Note that packets sent across layers experience different delays and conducting measurement in kernel suffers less delay. Besides, the popular monitoring mechanisms also introduce obvious delay to the measurement results. Note due to its close relation with Wifi state transition, the power management experiment is put in the sub-chapter below.

5.6.4 Effect of Factors in Category 3

We run MobiScope to perform ICMP-based RTT measurement at the system layer with different time intervals for estimating the delay due to the NIC state transition.

Delay due to cellular state transition. As shown in Figure 5.10, when the smartphone connects to 3G network, the measured RTTs are substantially inflated when the measurement interval exceeds 10s. Since there is no background traffic generated during the measurement period, we can infer that the RRC state switches from DCH to IDLE when the idle time exceeds 10s. Similarly, we can learn that the RRC transition from IDLE to DCH consumes around 700 ms.

When the smartphone connects to an LTE network, Figure 5.10 illustrates that the measured RTTs are increased by 200ms when the measurement intervals (i.e., idle

time) become larger than 15s. Therefore, we can infer that the timeout value of the CONNECTED state is 15s and the delay introduced by state switch from IDLE to CONNECTED is more than 200ms. Figure 5.10 also shows that the state transition of either LTE or 3G can cause significant delay to the measurement results, and the delay resulted from 3G state transition is more than three times of the delay brought by LTE state transition.

Delay due to WiFi state transition. We describe the result of sending packets and that of receiving packets, individually.

Packet sending. Figure 5.11 shows the measured RTT with different measurement intervals. For Samsung S3, the RTT values roughly center around 2ms, 6ms and 10ms, which are separated at intervals of 50ms and 200ms. For Nexus 5, the RTT values roughly center around 2ms, 5ms, 8ms, and 11ms. Moreover, they stay almost constant within a range of measurement intervals, and abruptly increase when the interval reaches a certain value (i.e., 50ms, 200ms, and 250ms). Both Samsung S3 and Nexus 5 devices have a state transition when the measurement interval is 200ms.

By analyzing the captured 802.11 frames, we find that after 200ms of inactivity (i.e., *ListenInterval*=200ms) the WiFi interface goes to sleep. Therefore, if the measurement interval exceeds 200ms, the WiFi interface goes to sleep after finishing an RTT measurement and then wakes up for the next measurement. Before conducting a new measurement after waking up, the WiFi interface needs some time to send a null data frame (NDF) with power management bit set to 0 to retrieve frames buffered at the AP. We can see that such additional time does not exist when measurement intervals are less than 200ms. Moreover, the jumps at 50ms for Samsung S3 and both 50ms and 250ms for Nexus 5 suggest other state transitions. Due to the lack of the WiFi driver's source code, we could just conjecture that they have proprietary PSM mechanisms, and we will

investigate on it in future work.

For TCP ping, the tools `Ping&DNS` and `PingTools` adopt T-P2 implementation pattern and `WiFi Speed Test` is implemented based on T-P1 pattern. As we represent the web server in the test bed using IP address instead of domain name, the result of these three tools are $t_3 - t_2$ as shown in Figure 5.1.

Packet receiving. Figure 5.12 shows the RTT values when the screen is on and off, respectively. By analyzing the captured 802.11 frames, we find that the WiFi interface adopts two different PSM schemes when the screen is on and off. Specifically, when the screen is on, the WiFi interface wakes up with a time interval of 200ms to check buffered packets in the AP. When the screen is off, such a time interval is 600ms.

Summary The NIC (i.e., 3G, LTE, WiFi) state transition can introduce obvious delays to the measurement. LTE may cause less delay than 3G, and different WiFi chipsets have different effects. Moreover, the effects are not the same when the smartphone is at different status (i.e., screen on/off).

5.6.5 Evaluation of `kping` and `kband`

In this experiment, we connect the smartphone to a host via USB tethering and limit the bandwidth to 100Mbps. Setting the server as the destination, we run `kping` and `ping` to measure the RTT. Figure 5.13(a) shows that the RTTs measured by `kping` center around 0.3ms with an upper bound of 0.4ms whereas the RTTs measured by `ping` are highly dispersed and fluctuated between 0.3ms and 1.3ms.

To evaluate the accuracy of capacity measurement, we run `kband` and `iperf` (native program) on the smartphone, both of which send 1498-byte UDP packets to the server for measurement. Moreover, to generate cross traffic, we run D-ITG [6] in the smartphone,

which sends UDP packets through raw socket at the system layer. The rate of cross traffic ranges from 0pkt/s to 8000pkt/s with an incremental step of 2000pkt/s. Figure 5.13(b) shows the capacity measured by `iperf` and `kband` with varying cross traffic from the smartphone to the server. We observe that when the volume of the cross traffic increases both `iperf`'s and `kband`'s accuracy of capacity measurement decreases. However, compared with `iperf`, `kband` is robust to cross traffic when measuring capacity. For example, when the cross traffic reaches 8000pkt/s, `kband` can still achieve a high accuracy with only 4.54% underestimation whereas `iperf` underestimates the capacity up to 49.54%.

Summary Conducting measurement in the kernel layer can obtain more accurate and stable results. Moreover, it is more robust to cross traffic than the measurement at above layers.

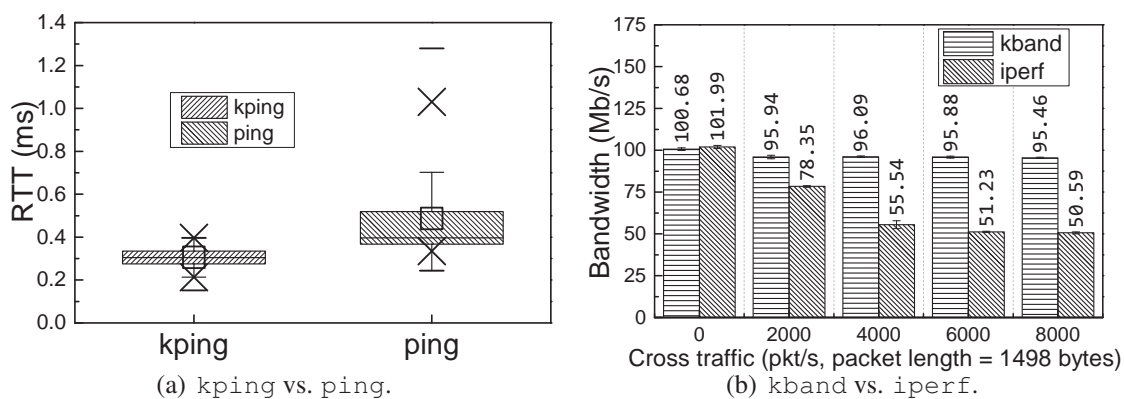


Figure 5.13: Evaluation of `kping` and `kband`.

5.7 Examining Descriptions of Measurement Apps

Since ambiguous descriptions may confuse users what is measured, we construct enhanced descriptions, each of which comprises a statement based on the static and dynamic analysis and a figure showing how the measurement is performed. To evaluate the clarity of the

official description and the enhanced one, we design questionnaires for 10 measurement apps [14] and conduct user studies.

5.7.1 Questionnaire Design

Figure 5.14 shows a snippet of the questionnaire of a particular app. The figure shows how the app actually conducts the measurement. The official description and the enhanced one are presented in random orders unknown to respondents. In other words, DESC1/DESC2 could be the official one or the enhanced one. Such randomness is to avoid biased result due to direct or indirect exposure of the official description. Note that the official description is fetched from Google Play. For each description, we ask respondents whether it is clear. They can only select one answer from “Abs YES”, “YES”, “Neutral”, “NO”, “Abs NO”, resulting in integer scores from five to one, respectively. We recruit 29 respondents with knowledge of computer networks from three cities because without such knowledge, a respondent may not understand the descriptions.

Apps	Procedure of the ping task	DESC1	Is DESC1 clear?	DESC2	Is DESC2 clear?
Ping&DNS		Ping a server (via ICMP over IPv4 or IPv6 and TCP), DNS lookup (with geographical lookup of IP addresses)	<input type="checkbox"/> Abs YES <input type="checkbox"/> YES <input type="checkbox"/> Neutral <input type="checkbox"/> NO <input type="checkbox"/> Abs NO	Ping a server using TCP packets and the RTT result contains the time consumed to perform one DNS lookup and build one TCP connection	<input type="checkbox"/> Abs YES <input type="checkbox"/> YES <input type="checkbox"/> Neutral <input type="checkbox"/> NO <input type="checkbox"/> Abs NO

Figure 5.14: A snippet of the questionnaire.

5.7.2 Result Analysis

We first investigate the clarity of official descriptions. For the 290 answers (10 answers in each questionnaire), 25.8% (i.e., (13+62)/290) of them are “Abs YES”/“YES” (i.e., the

respondents think that the official description is clear). 7.9% (i.e., 23/290) and 27.9% (i.e., 81/290) of them are “Abs NO” and “NO”, respectively. For the enhanced descriptions, 71.7% (i.e., (97+111)/290) of the answers are “Abs YES”/“YES” and only 5.2% (i.e., (5+10)/290) of them are “Abs NO”/“NO”.

We contrast the official description with the enhanced one derived from static and dynamic analysis. We find that the description of `Httping Tool` contradicts its implementation because it claims to measure HTTP latency, but its implementation measures the latency for establishing a TCP connection. The descriptions of most apps are ambiguous due to limited details. For example, `PingTools`'s description only has a simple sentence “ICMP, TCP and HTTP ping” without any details. Note that the result of its HTTP ping could be *twice* of the result of its TCP ping, because it adopts **H-P2** for HTTP ping and **T-P2** for TCP ping. Such a difference will confuse users. The description of `HTTPing`, in our opinion, is the most accurate and complete. Although `Netalyzr`'s description is simple, it clearly explains the meaning of each measurement result. That is also a good practice.

Chapter 6

Conclusion

6.1 Conclusion

Android has become the most popular mobile OS, and not all Android apps are benign and well designed. Besides, static analysis could be easily impeded by the dynamic features of programming languages or the protection mechanisms. In this thesis, we focus on using dynamic analysis to identify the apps' malicious actions and locate their potential performance issues, and we have proposed a series of efficient dynamic analysis approaches to monitor the behaviours of the target apps.

First, to assist Android security experts in analyzing potential malicious actions of Android apps in various layers, we propose a novel on-device non-invasive analysis system named Malton for the new Android runtime (i.e., ART). It provides a comprehensive view of the Android malware behaviours, by conducting multi-layer monitoring and tracking, as well as efficient path exploration. We have developed a prototype of Malton and the evaluation with sophisticated real-world malware samples demonstrated the effectiveness of our system.

Second, more and more malware authors start to employ packing techniques to hide

the malicious code and impede static analysis, and the evolving app packers can easily circumvent existing unpackers because they adopt a one-pass strategy and cannot monitor the behaviours of the packers in multiple layers, and hence such unpacking tools are not adaptive to the changes of packers. To address this challenging issue, we propose a novel iterative process and develop *PackerGrind* to recover the Dex files from packed apps. With the capability of conducting cross-layer tracking in real smartphones, *PackerGrind* can effectively monitor the packing patterns and adapt to the evolution of packers for extracting Dex files. Our extensive experiments with real packed apps illustrate the effectiveness and efficiency of *PackerGrind*.

Moreover, we employ a cross-layer analysis methodology to locate the potential issues that affect the measurement results of network performance. We conduct the *first* systematic investigation on why the measurement result from apps may be not what users have expected. First, we identify new factors, revisit known factors, and propose a novel approach as well as two tools to discover these factors in proprietary apps. Moreover, we perform extensive experiments to quantify the negative effects of these factors, and develop *MobiScope* for demonstrating how to mitigate such effects. Second, we find that the measurement apps' descriptions may be ambiguous and perplexing to its users. We construct enhanced descriptions to provide users more information on what is measured. The user studies show the improvement of the enhanced descriptions. This research sheds light on creating better mobile measurement apps and conducting expected network measurement in apps.

6.2 Future Work

As we have done three major works related to cross-layer analysis of Android apps in this thesis, future work also lies in the following directions.

First, both Malton and *PackerGrind* are based on the Valgrind framework. Similar to the anti-emulator techniques, malware could detect the existences of Valgrind and then stop executing malicious payloads. For example, the malware could check the app starting command or the time used to finish some operations. To address this challenge, for Malton we could leverage the path exploration mechanism to explore code paths and trigger conditionally executed payloads. For *PackerGrind*, we could change the return value of selected APIs to hide the existence of *PackerGrind*, or insert additional IR statements to modify the registers and force the app to execute forward. Nevertheless, it is an arms race between the analysis tools and anti-analysis techniques.

Though the in-memory optimization of Malton significantly reduces the code required to be executed, it is semi-automated because the entry point and the exit point of the interested code region need to be specified by analysts. How to fully automate this process is an interesting research direction that we will pursue. Besides, the direct execution needs analysts to specify which branches should be executed directly. In our current prototype, it simply ignores all possible crashes because the directly executed code path may access invalid memory locations. Hence, advanced malware may be able to evade it by exploiting this weakness. We will borrow some ideas from the X-Force [133] system in future work to recover execution from crashes automatically.

Moreover, *PackerGrind* currently focuses on the operations related to Dex files. Packed apps may hide the real code by modifying the compiled code in Oat files directly. Moreover, if the packed apps load different code into the same memory and execute them under different conditions, *PackerGrind* cannot decide which code is real. Since *PackerGrind* can trace such modifications and monitor code execution, we will address these issues by employing more semantic information in future work.

In addition, the code coverage is a concern for all dynamic analysis platforms, including

Malton and *PackerGrind*. For Malton, we leverage the monkey tool to generate events, and use the path exploration engine to explore different code paths. Even using the simple monkey tool, Malton has demonstrated better results than the existing tools as shown in Chapter 3.4.1. In future work, we will equip Malton with UI automation frameworks (e.g., [97]) to generate more directive events. *PackerGrind* can only recover the Dex data after the methods of releasing the real code are invoked. The majority of the existing packers execute such methods, which are usually JNI methods, when a packed app is launched to avoid performance degradation. We also use the mechanisms of IntelliDroid [172] and SmartGen [200] to trigger the execution of such methods. Since packers may delay the execution of such methods after knowing the mechanism of *PackerGrind*, we will leverage advanced input generator for Android [67, 121] to enhance *PackerGrind* in future work. Also, Malton uses taint analysis to track sensitive information propagation. But it cannot track implicit information flow. We will enhance it by leveraging the ideas in systems handling implicit information flows [104].

In addition, although we have employed the cross-layer analysis mechanism to identify the impactors that affect the results of mobile measurement apps, we will use this mechanism to address more challenging issues, such as power management, in future work.

Bibliography

- [1] Android developer reference. <https://goo.gl/p3sMxA>.
- [2] Android-unpacker. <https://github.com/strazzere/android-unpacker>.
- [3] Apk protect. <https://sourceforge.net/projects/apkprotect>.
- [4] ART and Dalvik. <https://source.android.com/devices/tech/dalvik/>.
- [5] Cloc: Count lines of code. <http://cloc.sourceforge.net/>.
- [6] D-itg. <http://traffic.comics.unina.it/software/ITG/>.
- [7] Dalvik bytecode. <https://source.android.com/devices/tech/dalvik/dalvik-bytecode.html>.
- [8] The history of xxshenqi and the future of sms phishing. <https://goo.gl/6Ds8NF>.
- [9] Java-basic datatypes. http://www.tutorialspoint.com/java/java_basic_datatypes.htm.
- [10] Keeping the device awake. <https://goo.gl/Kb0QZE>.
- [11] LG Nexus 5. http://www.gsmarena.com/lg_nexus_5-5705.php.
- [12] logcat command-line tool. <https://developer.android.com/studio/command-line/logcat.html>.
- [13] ltrace. <https://ltrace.org/>.

- [14] The measurement tools under evaluation. <https://goo.gl/MGtoAO>.
- [15] Profiling with traceview and dmtracedump. <http://goo.gl/QZRXEW>.
- [16] Soot. <http://sable.github.io/soot/>.
- [17] strace. <http://sourceforge.net/projects/strace>.
- [18] Sysfs. <https://www.kernel.org/doc/Documentation/filesystems/sysfs.txt>.
- [19] Vpnservice. <https://goo.gl/2tqRTm>.
- [20] Qemu. http://wiki.qemu.org/Main_Page, 2013.
- [21] Measuring broadband america mobile broadband services. <https://goo.gl/3MmsSP>, 2014.
- [22] OWASP mobile top 10 risks. https://www.owasp.org/index.php/OWASP_Mobile_Security_Project#tab=Top_10_Mobile_Risks, 2014.
- [23] F-Droid. <https://f-droid.org/>, 2015.
- [24] Baksmali. <https://github.com/JesusFreke/smali>, 2016.
- [25] CF-Bench. <http://http://bench.chainfire.eu/>, 2016.
- [26] Dalvik executable format. <https://source.android.com/devices/tech/dalvik/dex-format.html>, 2016.
- [27] Dex2jar. <https://github.com/pxb1988/dex2jar>, 2016.
- [28] Dexdump. <https://developer.android.com/studio/command-line/index.html>, 2016.
- [29] IDA Pro. <https://www.hex-rays.com/products/ida/>, 2016.
- [30] Jadx. <https://github.com/skylot/jadx>, 2016.

- [31] Platform versions dashboards. <https://goo.gl/YRW9II>, July 2016.
- [32] ZjDroid. <https://github.com/halfkiss/ZjDroid>, 2016.
- [33] 99.6 percent of new smartphones run android or ios. <https://www.theverge.com/2017/2/16/14634656/android-ios-market-share-blackberry-2016>, February 2017.
- [34] Number of available android applications. <https://www.appbrain.com/stats/number-of-android-apps>, June 2017.
- [35] Adrien Abraham, Radoniaina Andriatsimandefitra, Adrien Brunelat, J-F Lalande, and V Viet Triem Tong. Grodddroid: a gorilla for triggering malicious behaviors. In *Proc. MALWARE*, 2015.
- [36] Vitor Afonso, Antonio dBianchi, Yanick Fratantonio, Adam Doupé, Mario Polino, Paulo de Geus, Christopher Kruegel, and Giovanni Vigna. Going native: Using a large-scale analysis of android apps to create a practical native-code sandboxing policy. In *NDSS*, 2016.
- [37] Vitor Monte Afonso, Matheus Favero de Amorim, André Ricardo Abed Grégio, Glaucio Barroso Junquera, and Paulo Lício de Geus. Identifying android malware using dynamically obtained features. *Journal of Computer Virology and Hacking Techniques*, 2015.
- [38] Shaun Aimoto. Five ways Android malware is becoming more resilient. <https://www.symantec.com/connect/blogs/five-ways-android-malware-becoming-more-resilient>, 2016.
- [39] Shunya akamoto, Kenji Okuda, Ryo Nakatsuka, and Toshihiro Yamauchi. Droidtrack: tracking and visualizing information diffusion for preventing information leakage on android. *Journal of Internet Services and Information Security*, 2014.
- [40] Shahid Alam, Zhengyang Qu, Ryan Riley, Yan Chen, and Vaibhav Rastogi. Droidnative: Semantic-based detection of android native code malware. *arXiv preprint arXiv:1602.04693*, 2016.
- [41] Alibaba Inc. <http://jaq.alibaba.com/>.

- [42] Guy Almes, Matthew J Zekauskas, and Sunil Kalidindi. Rfc 2681: A round-trip delay metric for ippm, Sept. 1999.
- [43] Mohammed Alzaylaee, Suleiman Yerima, and Sakir Sezer. Emulator vs real phone: Android malware detection using machine learning. In *Proc. ACM IWSPA*, 2017.
- [44] Saswat Anand, Mayur Naik, Mary Jean Harrold, and Hongseok Yang. Automated concolic testing of smartphone apps. In *Proc. ACM FSE*, 2012.
- [45] Saswat Anand, Corina S Păsăreanu, and Willem Visser. Jpf-cse: A symbolic execution extension to java pathfinder. In *Proc. TACAS*, 2007.
- [46] Radoniaina Andriatsimandefitra and Valérie Viet Triem Tong. Capturing android malware behaviour using system flow graph. In *Proc. NSS*, 2014.
- [47] Axelle Apvrille and Ruchna Nigam. Obfuscation in Android malware, and how to fight back. *Virus Bulletin*, July 2014.
- [48] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *ACM SIGPLAN Notices*, 2014.
- [49] Elias Athanasopoulos, Vasileios Kemerlis, Georgios Portokalidis, and Angelos Keromytis. Naclndroid: Native code isolation for android applications. In *Proc. ESORICS*, 2016.
- [50] Kathy Au, Yi Fan Zhou, Zhen Huang, and David Lie. PScout: Analyzing the Android permission specification. In *Proc. CCS*, 2012.
- [51] Vitalii Avdiienko, Konstantin Kuznetsov, Alessandra Gorla, Andreas Zeller, Steven Arzt, Siegfried Rasthofer, and Eric Bodden. Mining apps for abnormal usage of sensitive data. In *Proc. ACM/IEEE ICSE*, 2015.
- [52] Michael Backes, Sven Bugiel, Christian Hammer, Oliver Schranz, and Philipp von Styp-Rekowsky. Boxify: Full-fledged app sandboxing for stock android. In *Proc. USENIX Security*, 2015.
- [53] Michael Backes, Sven Bugiel, Oliver Schranz, Philipp von Styp-Rekowsky, and Sebastian Weisgerber. Artist: The android runtime instrumentation and security toolkit. *arXiv preprint arXiv:1607.06619*, 2016.

- [54] Michael Backes, Sebastian Gerling, Christian Hammer, Matteo Maffei, and Philipp von Styp-Rekowsky. Appguardcenforcing user requirements on android apps. In *Proc. TACAS*, 2013.
- [55] Baidu Inc. <http://app.baidu.com>.
- [56] Bangle Inc. <http://www.bangle.com/>.
- [57] Paulo Barros, René Just, Suzanne Millstein, Paul Vines, Werner Dietl, Michael D Ernst, et al. Static analysis of implicit control flow: Resolving java reflection and android intents. In *Proc. IEEE/ACM ASE*, 2015.
- [58] Pascal Berthome, Thomas Fecherolle, Nicolas Guilloteau, and Jean-Francois Lalande. Repackaging android applications for auditing access to private data. In *Proc. ARES*, 2012.
- [59] Antonio Bianchi, Yanick Fratantonio, Christopher Kruegel, and Giovanni Vigna. Njas: Sandboxing unmodified applications in non-rooted devices running stock android. In *Workshop SPSM*, 2015.
- [60] Guillaume Bonfante, Jose Fernandez, Jean-Yves Marion, Benjamin Rouxel, Fabrice Sabatier, and Aurélien Thierry. CoDisasm: Medium scale concatic disassembly of self-modifying binaries with overlapping instructions. In *Proc. CCS*, 2015.
- [61] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proc. OSDI*, 2008.
- [62] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. Unleashing mayhem on binary code. In *Proc. IEEE SP*, 2012.
- [63] Kai Chen, Peng Wang, Yeonjoon Lee, XiaoFeng Wang, Nan Zhang, Heqing Huang, Wei Zou, and Peng Liu. Finding unknown malice in 10 seconds: Mass vetting for new threats at the Google-Play scale. In *Proc. USENIX Security*, 2015.
- [64] Qi Chen, Haokun Luo, Sanae Rosen, Z. Mao, Karthik Iyer, Jie Hui, Kranthi Sontineni, and Kevin Lau. Qoe doctor: Diagnosing mobile app qoe with automated ui control and cross-layer analysis. In *Proc. ACM IMC*, 2014.

- [65] Ting Chen, Xiao-Song Zhang, Xiao-Li Ji, Cong Zhu, Yang Bai, and Yue Wu. Test generation for embedded executables via concolic execution in a real environment. *IEEE Transactions on Reliability*, 2015.
- [66] Ting Chen, Xiaosong Zhang, Shize Guo, Hongyuan Li, and Yue Wu. State of the art: Dynamic symbolic execution for automated test generation. *Future Generation Computer Systems*, 29(7), 2013.
- [67] Shaunik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. Automated test input generation for android: Are we there yet? In *Proc. IEEE/ACM ASE*, 2015.
- [68] Mihai Christodorescu and Somesh Jha. Static analysis of executables to detect malicious patterns. Technical report, DTIC Document, 2006.
- [69] CISCO. Cisco 2014 annual security report. <https://www.cisco.com/web/offers/lp/2014-annual-security-report/index.html>, 2014.
- [70] Christian Collberg and Jasvir Nagra. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Addison-Wesley, 2009.
- [71] Mauro Conti, Vu Thien Nga Nguyen, and Bruno Crispo. Crepe: Context-related policy enforcement for android. In *Proc. ICIS*, 2010.
- [72] IDC Corporate. Worldwide smartphone market will see the first single-digit growth year on record. <https://goo.gl/1bJ1lt>, Dec. 2015.
- [73] Valerio Costamagna and Cong Zheng. Artdroid: A virtual-method hooking framework on android art runtime. In *Proc. ESSoS Workshop IMPS*, 2016.
- [74] Shuaifu Dai, Tao Wei, and Wei Zou. Droidlogger: Reveal suspicious behavior of android applications via instrumentation. In *Proc. ICCCT*, 2012.
- [75] Benjamin Davis, Ben Sanders, Armen Khodaverdian, and Hao Chen. I-arm-droid: A rewriting framework for in-app reference monitors for android applications. *Mobile Security Technologies*, 2012.
- [76] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proc. TACAS*, 2008.

- [77] Shuo Deng and Hari Balakrishnan. Traffic-aware techniques to reduce 3g/lte wireless energy consumption. In *Proc. ACM CoNEXT*, 2012.
- [78] Anthony Desnos and Geoffroy Gueguen. Androguard. <https://code.google.com/archive/p/droidbox>.
- [79] Anthony Desnos and Patrik Lantz. Droidbox: An android application sandbox for dynamic analysis. <https://code.google.com/archive/p/droidbox>, 2014.
- [80] Gianluca Dini, Fabio Martinelli, Andrea Saracino, and Daniele Sgandurra. MADAM: a multi-level anomaly detector for Android malware. In *Proc. MMM-ACNS*, 2012.
- [81] Dent A Earl et al. Structure harvester: a website and program for visualizing structure output and implementing the evanno method. *Conservation genetics resources*, 2012.
- [82] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung Gon Chun, Landon Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems*, 2014.
- [83] William Enck, Damien Ocateau, Patrick McDaniel, and Swarat Chaudhuri. A study of android application security. In *Proc. USENIX security*, 2011.
- [84] Hossein Falaki, Dimitrios Lymberopoulos, Ratul Mahajan, Srikanth Kandula, and Deborah Estrin. A first look at traffic on smartphones. In *Proc. ACM IMC*, 2010.
- [85] Ming Fan, Jun Liu, Xiapu Luo, Kai Chen, Tianyi Chen, Zhenzhou Tian, Xiaodong Zhang, and Ting Liu. Frequent subgraph based familial classification of Android malware. In *Proc. ISSRE*, 2016.
- [86] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android permissions demystified. In *Proc. CCS*, 2011.
- [87] Adrienne Porter Felt, Helen J Wang, Alexander Moshchuk, Steve Hanna, and Erika Chin. Permission re-delegation: Attacks and defenses. In *Proc. USENIX Security*, 2011.

- [88] Yanick Fratantonio, Antonio Bianchi, William Robertson, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Triggerscope: Towards detecting logic bombs in android applications. In *Proc. IEEE SP*, 2016.
- [89] Gartner, Inc. Debunking six myths of app wrapping. <https://www.gartner.com/doc/3008117/debunking-myths-app-wrapping>, 2015.
- [90] Alexandre Gerber, Subhabrata Sen, and Oliver Spatscheck. A call for more energy-efficient apps. *AT&T Labs Research*, 2011.
- [91] Utkarsh Goel, Mike P Wittie, Kimberly C Claffy, and Andrew Le. Survey of end-to-end mobile network measurement testbeds, tools, and services. *IEEE Communications Surveys Tutorials*, 2016.
- [92] Michael Grace, Yajin Zhou, Qiang Zhang, Shihong Zou, and Xuxian Jiang. Riskranker: Scalable and accurate zero-day android malware detection. In *Proc. MobiSys*, 2012.
- [93] Michael C Grace, Wu Zhou, Xuxian Jiang, and Ahmad Reza Sadeghi. Unsafe exposure analysis of mobile in-app advertisements. In *Proc. WiSec*, 2012.
- [94] Michael C Grace, Yajin Zhou, Zhi Wang, and Xuxian Jiang. Systematic detection of capability leaks in stock android smartphones. In *Proc. NDSS*, 2012.
- [95] Fanglu Guo, Peter Ferrie, and Tzi-Cker Chiueh. A study of the packer problem and its solutions. In *Proc. RAID*, 2008.
- [96] Hao Han, Yunxin Liu, Guobin Shen, Yongguang Zhang, and Qun Li. Dozyap: power-efficient wi-fi tethering. In *Proc. ACM MobiSys*, page 2012, 2012.
- [97] Shuai Hao, Bin Liu, Suman Nath, William G.J. Halfond, and Ramesh Govindan. Puma: Programmable ui-automation for large-scale dynamic analysis of mobile apps. In *Proc. MobiSys*, 2014.
- [98] Amir Houmansadr, Saman A Zonouz, and Robin Berthier. A cloud-based intrusion detection and response system for mobile phones. In *Proc. IEEE DSN-W*, 2011.
- [99] Junxian Huang, Qiang Xu, Birjodh Tiwana, Z Morley Mao, Ming Zhang, and Paramvir Bahl. Anatomizing application performance differences on smartphones. In *Proc. ACM MobiSys*, 2010.

- [100] Ijiami Inc. <http://www.ijiami.cn/>.
- [101] Cheol Jeon, WooChur Kim, Bongjae Kim, and Yookun Cho. Enhancing security enforcement on unmodified android. In *Proc. SAC*, 2013.
- [102] Jesusfreke. smali. <https://code.google.com/p/smali/>.
- [103] Yiming Jing, Ziming Zhao, Gail-Joon Ahn, and Hongxin Hu. Morpheus: automatically generating heuristics to detect android emulators. In *Proc. ACM ACSAC*, 2014.
- [104] Min Gyung Kang, Stephen McCamant, Pongsin Poosankam, and Dawn Song. Dta++: Dynamic taint analysis with targeted control-flow propagation. In *Proc. NDSS*, 2011.
- [105] Mohammad Karami, Mohamed Elsabagh, Parnian Najafiborazjani, and Angelos Stavrou. Behavioral analysis of android applications using automated instrumentation. In *Proc. SERE-C*, 2013.
- [106] Gordon Kelly. 97% of mobile malware is on Android. this is the easy way you stay safe. <http://arcy24.blogspot.hk/2014/09/report-97-of-mobile-malware-is-on.html>, 2014.
- [107] Dongwoo Kim, Jin Kwak, and Jaecheol Ryou. DWroidDump: Executable code extraction from Android applications for malware analysis. *International Journal of Distributed Sensor Networks*, 2015.
- [108] Dhilung Kirat and Giovanni Vigna. Malgene: Automatic extraction of malware analysis evasion signature. In *Proc. ACM CCS*, 2015.
- [109] Christian Kreibich, Nicholas Weaver, Boris Nechaev, and Vern Paxson. Netalyzer: Illuminating the edge network. In *Proc. ACM IMC*, 2010.
- [110] Andrea Lanzi, Monirul I Sharif, and Wenke Lee. K-tracer: A system for extracting kernel malware behavior. In *proc. NDSS*, 2009.
- [111] Seokjun Lee, Chanmin Yoon, and Hojung Cha. User interaction-based profiling system for android application tuning. In *Proc. Ubicomp*, 2014.
- [112] Li Li, Alexandre Bartel, Tegawendé F. Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Ochteau, and Patrick

- McDaniel. IccTA: Detecting inter-component privacy leaks in Android apps. In *Proc. ICSE*, 2015.
- [113] Li Li, Alexandre Bartel, Tegawendé François D Assise Bissyande, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Ochteau, and Patrick McDaniel. Iccta: detecting inter-component privacy leaks in android apps. In *Proc. ICSE*, 2015.
- [114] Weichao Li, Ricky Mok, Daoyuan Wu, and Rocky Chang. On the accuracy of smartphone-based mobile network measurement. In *Proc. IEEE INFOCOM*, 2015.
- [115] Ying-Dar Lin, Yuan-Cheng Lai, Chien-Hung Chen, and Hao-Chuan Tsai. Identifying android malicious repackaged applications by thread-grained system call sequences. *computers & security*, 2013.
- [116] Martina Lindorfer, Matthias Neugschwandtner, and Christian Platzer. Marvin: Efficient and comprehensive mobile app classification through static and dynamic analysis. In *Proc. COMPSAC*, 2015.
- [117] Martina Lindorfer, Matthias Neugschwandtner, Lukas Weichselbaum, Yanick Fratantonio, Victor Van Der Veen, and Christian Platzer. Andrubis—1,000,000 apps later: A view on current android malware behaviors. In *Workshop BADGERS*, 2014.
- [118] Chi Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. *ACM Sigplan Notices*, 40(6), 2005.
- [119] Riyadh Mahmood, Nariman Mirzaei, and Sam Malek. Evodroid: Segmented evolutionary testing of android apps. In *Proc. FSE*, 2014.
- [120] Sam Malek, Naeem Esfahani, Thabet Kacem, Riyadh Mahmood, Nariman Mirzaei, and Angelos Stavrou. A framework for automated security testing of android applications on the cloud. In *Proc. IEEE SERE-C*, 2012.
- [121] Nariman Mirzaei, Joshua Garcia, Hamid Bagheri, Alireza Sadeghi, and Sam Malek. Reducing combinatorics in gui testing of Android applications. In *Proc. ACM/IEEE ICSE*, 2016.

- [122] Al Morton. Rfc 7497: Rate measurement test protocol problem statement and requirements, Apr. 2015.
- [123] Andreas Moser, Christopher Kruegel, and Engin Kirda. Limits of static analysis for malware detection. In *Proc. ACSAC*, 2007.
- [124] NAGA IN Inc. <http://www.nagain.com/>.
- [125] Annamalai Narayanan, Liu Yang, Lihui Chen, and Liu Jinliang. Adaptive and scalable android malware detection through online learning. In *Proc. IJCNN*, 2016.
- [126] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan notices*, 2007.
- [127] Netqin Inc. <https://www.netqin.com>.
- [128] Ashkan Nikraves, Hongyi Yao, Shichang Xu, David Choffnes, and Z Morley Mao. Mobilyzer: An open platform for controllable mobile network measurements. In *Proc. MobiSys*, 2015.
- [129] Palo Alto Networks. Wildfire[tm] cloud-based threat analysis service. <https://www.paloaltonetworks.com/products/secure-the-network/subscriptions/wildfire>.
- [130] Xuerui Pan, Yibing Zhongyang, Zhi Xin, Bing Mao, and Hao Huang. Defensor: Lightweight and efficient security-enhanced framework for android. In *Proc. TrustCom*, 2014.
- [131] Abhinav Pathak, Y. Charlie Hu, and Ming Zhang. Where is the energy spent inside my app?: Fine grained energy accounting on smartphones with eprof. In *Proc. EuroSys*, 2012.
- [132] PayEgis Inc. <http://www.payegis.com/>.
- [133] Fei Peng, Zhui Deng, Xiangyu Zhang, Dongyan Xu, Zhiqiang Lin, and Zhendong Su. X-force: Force-executing binary programs for security applications. In *Proc. USENIX Security*, 2014.
- [134] Thanasis Petsas, Giannis Voyatzis, Elias Athanasopoulos, Michalis Polychronakis, and Sotiris Ioannidis. Rage against the virtual machine: hindering dynamic analysis of Android malware. In *Proc. EuroSec*, 2014.

- [135] Andrew J Pyles, Xin Qi, Gang Zhou, Matthew Keally, and Xue Liu. Sapsm: Smart adaptive 802.11 psm for smartphones. In *Proc. ACM UbiComp*, 2012.
- [136] Chenxiong Qian, Xiapu Luo, Yu Le, and Guofei Gu. Vulhunter: Toward discovering vulnerabilities in android applications. *IEEE Micro*, 2015.
- [137] Chenxiong Qian, Xiapu Luo, Yuru Shao, and Alvin TS Chan. On tracking information flows through jni in android applications. In *Proc. IEEE/IFIP DSN*, 2014.
- [138] Feng Qian, Zhaoguang Wang, Alex Gerber, Z. Mao, Subhabrata Sen, and Oliver Spatscheck. Profiling resource usage for mobile applications: a cross-layer approach. In *Proc. ACM Mobisys*, 2011.
- [139] Qihoo360 Inc. <http://dev.360.cn/>.
- [140] Siegfried Rasthofer, Steven Arzt, Enrico Lovat, and Eric Bodden. Droidforce: enforcing complex, data-centric, system-wide policies in android. In *Proc. ARES*, 2014.
- [141] Siegfried Rasthofer, Steven Arzt, Marc Miltenberger, and Eric Bodden. Harvesting runtime values in android applications that feature anti-analysis techniques. In *Proc. NDSS*, 2016.
- [142] Vaibhav Rastogi, Yan Chen, and William Enck. Appsplayground: automatic security analysis of smartphone applications. In *Proc. CODASPY*, 2013.
- [143] Lenin Ravindranath, Suman Nath, Jitendra Padhye, and Hari Balakrishnan. Automatic and scalable fault detection for mobile applications. In *Proc. ACM MobiSys*, 2014.
- [144] Lenin Ravindranath, Jitendra Padhye, Sharad Agarwal, Ratul Mahajan, Ian Obermiller, and Shahin Shayandeh. Appinsight: Mobile app performance monitoring in the wild. In *Proc. OSDI*, 2012.
- [145] Lenin Ravindranath, Jitendra Padhye, Ratul Mahajan, and Hari Balakrishnan. Timecard: Controlling user-perceived delays in server-based mobile applications. In *Proc. ACM SOSP*, 2013.

- [146] Abbas Razaghpanah, Narseo Vallina-Rodriguez, Srikanth Sundaresan, Christian Kreibich, Phillipa Gill, Mark Allman, and Vern Paxson. Haystack: In situ mobile traffic analysis in user space. *CoRR, abs/1510.01419*, 2015.
- [147] Alessandro Reina, Aristide Fattori, and Lorenzo Cavallaro. A system call-centric analysis and stimulation technique to automatically reconstruct android malware behaviors. *Proc. EuroSec*, 2013.
- [148] Sanae Rosen, Haokun Luo, Qi Alfred Chen, Morley Mao, Jie Hui, Aaron Drake, and Kevin Lau. Discovering fine-grained rrc state dynamics and performance impacts in cellular networks. In *Proc. ACM MobiCom*, 2014.
- [149] Alireza Sadeghi, Hamid Bagheri, Joshua Garcia, and Sam Malek. A taxonomy and qualitative comparison of program analysis techniques for security assessment of android software. *IEEE Transactions on Software Engineering*, 2016.
- [150] Daniel Schreckling, Johannes Köstler, and Matthias Schaff. Kynoid: real-time enforcement of fine-grained, user-defined, and data-centric security policies for android. *Information Security Technical Report*, 2013.
- [151] Sebastian Schrittwieser, Stefan Katzenbeisser, Johannes Kinder, Georg Merzdovnik, and Edgar Weippl. Protecting software through obfuscation: Can it keep pace with progress in code analysis? *ACM Computing Surveys (CSUR)*, 2016.
- [152] Julian Schütte, Rafael Fedler, and Dennis Titze. Condroid: Targeted dynamic analysis of android applications. In *Proc. IEEE AINA*, 2015.
- [153] Julian Schütte, Dennis Titze, and JM De Fuentes. Appcaulk: Data leak prevention by injecting targeted taint tracking into android apps. In *Proc. TrustCom*, 2014.
- [154] Yuru Shao, Xiapu Luo, Chenxiong Qian, Pengfei Zhu, and Lei Zhang. Towards a scalable resource-driven approach for detecting repackaged android applications. In *Proc. ACSAC*, 2014.
- [155] Michael Spreitzenbarth, Felix Freiling, Florian Echtler, Thomas Schreck, and Johannes Hoffmann. Mobile-sandbox: having a deeper look into android applications. In *Proc.. SAC*, 2013.
- [156] Tim Strazzere and Jon Sawyer. Android hacker protection level 0. *DEFCON*, 2014.

- [157] Tzu Hsiang Su, Hsiang Jen Tsai, Keng Hao Yang, Po Chun Chang, Tien Fu Chen, and Yi Ting Zhao. Reconfigurable vertical profiling framework for the android runtime system. *ACM Transactions on Embedded Computing Systems (TECS)*, 13, 2014.
- [158] Li Sun, Ramanujan K Sheshadri, Wei Zheng, and Dimitrios Koutsonikolas. Modeling wifi active power/energy consumption in smartphones. In *Proc. IEEE ICDCS*, 2014.
- [159] Mengtao Sun and Gang Tan. Nativeguard: Protecting android applications from third-party native libraries. In *Proc. ACM WiSec*, 2014.
- [160] Mingshen Sun, John C. S. Lui, and Yajin Zhou. Blender: Self-randomizing address space layout for android apps. In *Proc. RAID*, 2016.
- [161] Mingshen Sun, Tao Wei, and John Lui. Taintart: A practical multi-level information-flow tracking system for android runtime. In *Proc. ACM CCS*, 2016.
- [162] Kimberly Tam, Ali Feizollah, Nor Badrul Anuar, Rosli Salleh, and Lorenzo Cavallaro. The evolution of android malware and android analysis techniques. *ACM Computing Surveys (CSUR)*, 49(4), 2017.
- [163] Kimberly Tam, Salahuddin J Khan, Aristide Fattori, and Lorenzo Cavallaro. CopperDroid: Automatic reconstruction of Android malware behaviors. In *Proc. NDSS*, 2015.
- [164] Tencent Inc. <https://www.qqcloud.com/product/cr>.
- [165] Xabier Ugarte-Pedrero, Davide Balzarotti, Igor Santos, and Pablo G Bringas. Sok: Deep packer inspection: A longitudinal study of the complexity of run-time packers. In *Proc. IEEE SP*, 2015.
- [166] Timothy Vidas and Nicolas Christin. Evading Android runtime analysis via sandbox detection. In *Proc. ASIACCS*, 2014.
- [167] David Wagner and Drew Dean. Intrusion detection via static analysis. In *Proc. IEEE SP*, 2001.
- [168] Xueqiang Wang, Kun Sun, Yuewu Wang, and Jiwu Jing. Deepdroid: Dynamically enforcing enterprise policy on android devices. In *Proc. NDSS*, 2015.

- [169] Xuetao Wei, Lorenzo Gomez, Iulian Neamtiu, and Michalis Faloutsos. Profiledroid: multi-layer profiling of android applications. In *Proc. ACM MobiCom*, 2012.
- [170] Zheng Wei and David Lie. Lazytainter: Memory-efficient taint tracking in managed runtimes. In *Proc. CCS-SPSM*, 2014.
- [171] Mike P Wittie, Brett Stone-Gross, Kevin C Almeroth, and Elizabeth M Belding. Mist: Cellular data network measurement for mobile applications. In *Proc. ICST BROADNETS*, 2007.
- [172] Michelle Y Wong and David Lie. IntelliDroid: A targeted input generator for the dynamic analysis of Android malware. In *Proc. NDSS*, 2016.
- [173] Wen-Chieh Wu and Shih-Hao Hung. Droiddolphin: a dynamic android malware detection framework using big data and machine learning. In *Proc. RACS*, 2014.
- [174] Mingyuan Xia, Lu Gong, Yuanhao Lyu, Zhengwei Qi, and Xue Liu. Effective real-time android application auditing. In *Proc. IEEE SP*, 2015.
- [175] Meng Xu, Chengyu Song, Yang Ji, Ming-Wei Shih, Kangjie Lu, Cong Zheng, Ruian Duan, Yeongjin Jang, Byoungyoung Lee, Chenxiong Qian, Sangho Lee, and Taesoo Kim. Toward engineering a secure android ecosystem: A survey of existing techniques. *ACM Computing Surveys (CSUR)*, 49(2), 2016.
- [176] Rubin Xu, Hassen Saïdi, and Ross J Anderson. Aurasium: Practical policy enforcement for android applications. In *Proc. USENIX Security*, 2012.
- [177] Lei Xue, Xiapu Luo, and Yuru Shao. ktrxr: A portable toolkit for reliable internet probing. In *Proc. IEEE IWQoS*, 2014.
- [178] Lei Xue, Xiapu Luo, Le Yu, Shuai Wang, and Dinghao Wu. Adaptive unpacking of android apps. In *Proc. ICSE*, 2017.
- [179] Lei Xue, Chenxiong Qian, and Xiapu Luo. Androidperf: A cross-layer profiling system for android applications. In *Proc. IEEE IWQoS*, 2015.
- [180] Lok-Kwong Yan and Heng Yin. DroidScope: Seamlessly reconstructing OS and Dalvik semantic views for dynamic Android malware analysis. In *Proc. USENIX Security*, 2012.

- [181] Chao Yang, Guangliang Yang, Ashish Gehani, Vinod Yegneswaran, Dawood Tariq, and Guofei Gu. Using provenance patterns to vet sensitive behaviors in Android apps. In *Proc. SecureComm*, 2015.
- [182] Wenbo Yang, Yuanyuan Zhang, Juanru Li, Junliang Shu, Bodong Li, Wenjun Hu, and Dawu Gu. AppSpear: Bytecode decrypting and DEX reassembling for packed Android malware. In *Proc. RAID*, 2015.
- [183] Zheming Yang, Min Yang, Yuan Zhang, Guofei Gu, Peng Ning, and X Sean Wang. Appintent: Analyzing sensitive data transmission in android for privacy leakage detection. In *Proc. ACM CCS*, 2013.
- [184] Hongyi Yao, Gyan Ranjan, Alok Tongaonkar, Yong Liao, and Zhuoqing Morley Mao. Samples: Self adaptive mining of persistent lexical snippets for classifying mobile application traffic. In *Proc. MobiCom*, 2015.
- [185] Chanmin Yoon, Dongwon Kim, Wonwoo Jung, Chulkoo Kang, and Hojung Cha. Appscope: application energy metering framework for android smartphones using kernel activity monitoring. In *Proc. USENIX ATC*, 2012.
- [186] Le Yu, Xiapu Luo, Xule Liu, and Tao Zhang. Can we trust the privacy policies of android apps? In *Proc. IEEE/IFIP DSN*, 2016.
- [187] Zhenlong Yuan, Yongqiang Lu, and Yibo Xue. Droiddetector: android malware characterization and detection using deep learning. *Tsinghua Science and Technology*, 21(1), 2016.
- [188] Matthew J Zekauskas, Anatoly Karp, Benjamin Teitelbaum, Stanislav Shalunov, and Jeff W Boote. Rfc 4656: A one-way active measurement protocol (owamp), Sept. 2006.
- [189] Lide Zhang, David Bild, Robert Dick, Zhuoqing Mao, and Peter Dinda. Panappticon: event-based tracing to measure mobile application and platform performance. In *Proc. CODES+ISSS*, 2013.
- [190] Mu Zhang, Yue Duan, Heng Yin, and Zhiruo Zhao. Semantics-aware android malware classification using weighted contextual api dependency graphs. In *Proc. ACM CCS*, 2014.
- [191] Mu Zhang and Heng Yin. Efficient, context-aware privacy leakage confinement for android applications without firmware modding. In *Proc. ASIACCS*, 2014.

- [192] Yuan Zhang, Min Yang, Bingquan Xu, Zhemin Yang, Guofei Gu, Peng Ning, X Sean Wang, and Binyu Zang. Vetting undesirable behaviors in android apps with permission use analysis. In *Proc. ACM CCS*, 2013.
- [193] Yueqian Zhang, Xiapu Luo, and Haoyang Yin. DexHunter: toward extracting hidden code from packed Android applications. In *Proc. ESORICS*, 2015.
- [194] Shuai Zhao, Xiaohong Li, Guangquan Xu, Lei Zhang, and Zhiyong Feng. Attack tree based android malware detection with hybrid analysis. In *Proc. TrustCom*, 2014.
- [195] Min Zheng, Mingshen Sun, and John CS Lui. DroidTrace: a ptrace based Android dynamic analysis system with forward execution capability. In *Proc. IWCMC*, 2014.
- [196] Wu Zhou, Yajin Zhou, Xuxian Jiang, and Peng Ning. DroidMOSS: Detecting repackaged smartphone applications in third-party android marketplaces. In *Proc. ACM CODASPY*, 2012.
- [197] Yajin Zhou and Xuxian Jiang. Dissecting Android malware: Characterization and evolution. In *Proc. IEEE SP*, 2012.
- [198] Yajin Zhou, Zhi Wang, Wu Zhou, and Xuxian Jiang. Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. In *Proc. NDSS*, 2012.
- [199] Saman Zonouz, Amir Houmansadr, Robin Berthier, Nikita Borisov, and William Sanders. Secloud: A cloud-based comprehensive and lightweight security solution for smartphones. *Computers & Security*, 37, 2013.
- [200] Chaoshun Zuo and Zhiqiang Lin. Smartgen: Exposing server urls of mobile apps with selective symbolic execution. In *Proc. WWW*, 2017.