

Singapore Management University

Institutional Knowledge at Singapore Management University

Dissertations and Theses Collection (Open Access)

Dissertations and Theses

7-2019

Sensitive behavior analysis of android applications on unrooted devices in the wild

Xiaoxiao TANG

Singapore Management University, xxtang.2013@phdis.smu.edu.sg

Follow this and additional works at: https://ink.library.smu.edu.sg/etd_coll



Part of the [Software Engineering Commons](#), and the [Systems Architecture Commons](#)

Citation

TANG, Xiaoxiao. Sensitive behavior analysis of android applications on unrooted devices in the wild. (2019). Dissertations and Theses Collection (Open Access).

Available at: https://ink.library.smu.edu.sg/etd_coll/222

This PhD Dissertation is brought to you for free and open access by the Dissertations and Theses at Institutional Knowledge at Singapore Management University. It has been accepted for inclusion in Dissertations and Theses Collection (Open Access) by an authorized administrator of Institutional Knowledge at Singapore Management University. For more information, please email cherylids@smu.edu.sg.

**SENSITIVE BEHAVIOR ANALYSIS OF ANDROID
APPLICATIONS ON UNROOTED DEVICES IN
THE WILD**

XIAOXIAO TANG

SINGAPORE MANAGEMENT UNIVERSITY

2019

Sensitive Behavior Analysis of Android Applications on Unrooted Devices in the Wild

by
Xiaoxiao TANG

Submitted to School of Information Systems in partial fulfillment of the
requirements for the Degree of Doctor of Philosophy in Information Systems

Dissertation Committee:

Debin GAO (Supervisor / Chair)
Associate Professor of Information Systems
Singapore Management University

Robert DENG Huijie (Co-supervisor)
Professor of Information Systems
Singapore Management University

Xuhua DING
Associate Professor of Information Systems
Singapore Management University

Zhenkai LIANG
Associate Professor
National University of Singapore

Singapore Management University
2019

Copyright (2019) Xiaoxiao TANG

“I hereby declare that this thesis / dissertation is my original work and it has been written by me in its entirety. I have duly acknowledged all the sources of information which have been used in this thesis / dissertation.

This thesis / dissertation has also not been submitted for any degree in any university previously.”

Tang

Xiaoxiao Tang

19 July 2019

Sensitive Behavior Analysis of Android Applications on Unrooted Devices in the Wild

Xiaoxiao TANG

Abstract

Dynamic analysis is widely used in malware detection, taint analysis, vulnerability detection, and other areas for enhancing the security of Android. Compared to static analysis, dynamic analysis is immune to common code obfuscation techniques and dynamic code loading. Existing dynamic analysis techniques rely on in-lab running environment (e.g., modified systems, rooted devices, or emulators) and require automatic input generators to execute the target app. However, these techniques could be bypassed by anti-analysis techniques that allow apps to hide sensitive behavior when an in-lab environment is detected through predefined heuristics (e.g., IMEI number of the device is invalid). Meanwhile, current input generators are still not intelligent enough to invoke adequate app behavior and provide sufficient code coverage. Therefore, it is an important research direction to investigate dynamic analysis techniques which enable a more complete execution under real running environments. This thesis focuses on dynamically analyzing app behavior by using public APIs and side-channel information, such that the techniques can be deployed on unrooted devices used by public users.

We first propose an advanced code obfuscation technique to hide small pieces of sensitive code with a code-reuse technique. This technique can hinder existing static analysis as well as dynamic analysis based on code-level events, such as API calls or Dalvik instructions. We implement a semi-automatic tool named AndroidCubo and show that it protects both Java and native code at a small runtime overhead.

Since relying on code-level event monitoring for revealing underlying app behavior can be bypassed by obfuscation and anti-analysis techniques, we propose a novel technique to dynamically monitoring apps by observing changes to public re-

sources on the device. We propose to observe the resources with public APIs and virtual file interfaces to monitor sensitive behavior, and then use machine learning techniques to identify the initiating app of the behavior. We implement a system named UpDroid which contains a monitor published on Google Play and a server-side analyzer. UpDroid can be easily deployed on devices used by the public and successfully monitor sensitive behavior of the app that is being analyzed. This work is a successful investigation of dynamic analysis on unrooted devices.

To conduct more fine-grained analysis on apps, we propose to use GPU interrupt timing information to infer the launched app and concrete behavior within a running app, such as layout switching. We obtain GPU interrupt timing information from a side channel - `/proc/interrupts`. We sample the number of the raised GPU interrupt and get the timing series while an activity occurs on the device to generate a feature vector for that activity. Then, we use machine learning techniques to train classification models for the activities. With the models, we are able to identify different types of app activities, e.g., identify the launched app or distinguish the activities within an app. This work further demonstrates the effectiveness of dynamic analysis on unrooted devices.

Finally, we conduct a simulation study for dynamically analyzing the factors that would affect the malware spreading on unrooted devices. In this work, we recruit participants to spread out messages, which simulates the malware spreading messages sent from infected mobile devices, to their friends. Each message contains a malicious-looking link to simulate the malware downloading links. When the participants spread out the messages, we use dynamic analysis to monitor the status of their devices and record the infection rate. The results show that spreading method, relationship, contact frequency would significantly affect the spreading of malware by analyzing the infection rates of different statuses of the device and the differences of the spreading messages.

Table of Contents

1	Introduction	1
2	Literature Review	5
2.1	Code Obfuscation Techniques	5
2.2	Dynamic Analysis on Android	7
2.3	Side channel and crowd sourcing	8
3	Obfuscation of Sensitive Behavior	10
3.1	Introduction	10
3.2	Overview	12
3.3	Turing Complete Gadget Set	13
3.4	Code Obfuscation	15
3.4.1	Essential Code Replacement	16
3.4.2	Payload Generation	18
3.4.3	Code Triggering	19
3.4.4	Payload Protection	20
3.5	Implementation and Case Studies	21
3.5.1	Implementation details	21
3.5.2	Case study: Obfuscating Native Code	23
3.5.3	Case study: Obfuscating Java Code	24
3.5.4	Overhead	25
3.6	Comparison with other Obfuscation Techniques	25

3.6.1	The Experiment	25
3.6.2	Reverse Engineering Results	26
3.6.3	Discussion	26
3.6.4	Limitations	28
3.7	Summary	28
4	Sensitive Behavior Analysis	29
4.1	Introduction	29
4.2	Background and Motivation	30
4.2.1	Resources and Observers	31
4.2.2	Motivation	32
4.3	System Overview	34
4.4	Event Monitoring	35
4.4.1	Content Observer	35
4.4.2	File Observer	36
4.4.3	Interrupt Observer	37
4.4.4	Network Observer	39
4.5	Initiator Identifying	39
4.5.1	App Status Monitoring	39
4.5.2	Data Collecting	41
4.5.3	Data Pre-processing	42
4.5.4	Modelling and Precision	43
4.6	Comparison with API hooking	44
4.6.1	Current State of API hooking	44
4.6.2	Permission Coverage Comparison	45
4.6.3	Event Details Comparison	47
4.6.4	Behavior Outcome Comparison	50
4.7	Capability Analysis	51
4.7.1	Permission Coverage	51

4.7.2	Runtime Experiments	53
4.7.3	Performance	54
4.7.4	Discussion	55
4.8	Summary	56
5	App Analysis with GPU Interrupt Timing Information	57
5.1	Introduction	57
5.2	Background	59
5.2.1	Interrupt Mechanism	59
5.2.2	GPU Interrupts	60
5.3	Methodology Overview	62
5.4	Experiments on Android	63
5.4.1	Experiments Setup	63
5.4.2	Model Precision	64
5.4.3	Noise Analysis	66
5.5	Experiments on Ubuntu	68
5.5.1	Experiments Setup	68
5.5.2	Model Precision	69
5.5.3	Noises Analysis	72
5.6	Discussion	73
5.7	Summary	73
6	Application of Dynamic Analysis in a Malware-Spreading Study	75
6.1	Introduction	75
6.2	Problem Statement	77
6.3	Simulation System	79
6.3.1	Overview	79
6.3.2	Seemingly malicious Message	80
6.3.3	Status Monitoring	80
6.4	Simulation Study	81

6.4.1	Recruitment	82
6.4.2	Seemingly malicious Message	83
6.4.3	Spreading	83
6.4.4	Results	84
6.5	Summary	88
7	Conclusion	89

List of Figures

3.1	Overview of code-reuse-based obfuscation on Android.	13
3.2	The native code of calling <code>sendTextMessage()</code> with JNI.	17
3.3	Layout of the payload.	19
3.4	Trigger code to be added to source code of the application.	20
3.5	Source code to be hidden and the corresponding gadget sequence.	23
3.6	Stack layout after loading the payload.	24
3.7	The decompiled code of function calls.	27
4.1	Framework of the sensitive behavior monitoring system - UpDroid.	34
4.2	Different categories of behavior that UpDroid can monitor	35
4.3	Overview of building the app identification model	40
4.4	The performance of different ranking algorithms in RankLib library.	44
4.5	Dangerous permission (covered by UpDroid) usage	52
4.6	Dangerous permission (not covered by UpDroid) usage of malware	52
4.7	The runtime analysis results of WhatsApp	53
4.8	Performance of UpDroid evaluated with Antutu Benchmark	54
5.1	GPU interrupt increasing patterns	59
5.2	A sample of the <code>/proc/interrupts</code> file on HUAWEI Nexus 6P	60
5.3	WhatsApp using scenarios that are difficult to be distinguished	65
5.4	Heatmap of inferring app on Android	66
5.5	Heatmap of inferring WhatsApp on Android	67
5.6	Heatmap of inferring webpage on Android	67

5.7	Precision of inferring app on Android	68
5.8	Heatmap of inferring app on Ubuntu	70
5.9	Heatmap of inferring webpage on Ubuntu	71
5.10	Precision of inferring app on Ubuntu	71
6.1	Malware spreading among mobile devices.	76
6.2	Overview of the system for simulation of malware spreading	79
6.3	Steps for users to customize the messages to be sent out	81
6.4	Number of clicks on the link	85
6.5	Infection rate of different spreading methods	85
6.6	Infection rate of different relationships	86
6.7	Infection rate of different time	87

List of Tables

3.1	Number of gadgets found in different gadget sets.	14
3.2	Number of different types of gadgets in our gadget set.	15
3.3	Examples of operations on Android	18
4.1	Existing tools for analyzing sensitive behavior of Android apps . . .	33
4.2	Features for running apps and the process combination rules.	42
4.3	The comparison of dangerous permission.	46
4.4	The comparison of normal permission.	48
4.5	The SMS event details.	49
5.1	The interrupt sources and description of NVIDIA GF119 [41] . . .	61
5.2	Precision of classifiers on Android	64
5.3	Precision of classifiers on Ubuntu	69

Acknowledgments

Throughout the writing of this dissertation, I received a great deal of support and assistance. First and foremost I would like to thank my PHD supervisor, Dr. Debin Gao, for his guidance, support, and patience. Dr. Gao's expertise was invaluable in the formulating of the research topic and methodology. I really want to thank Dr. Gao for the advice and support he gave me on the research of dynamic analysis and the patience he gave when I encountered difficulties in carrying out my work.

I also want to thank my lab-mates in SMU: Yan Lin, Xu Ke, Siqi Ma, Siqi Zhao, Daoyuan Wu, Ximin Liu, Jiaqi Hong, Jiayun Xu. I learned a lot from the insightful discussion of the research topic with them and I'm grateful to the generous help they gave me in my research. In particular, I want to thank Yan Lin for her help in the projects we cooperated.

I am also grateful to my tutor, Dr. Lujo Bauer, and my colleague, Haley Bui-Nguyen, during the CMU-LARC exchange program. I want to thank Dr. Bauer for the insights he gave to me in choosing research topics and Haley for her help in our cooperation.

At last, I would like to thank my family and friends for their support. I want to thank my parents, who support every decision I made and are always there when I need help. I also want to acknowledge my husband, Dr. Yaxiong Xie. I am grateful to the advice he provides about research, the experiences he shares in paper writing, and all the help he gives in life. It is because of his love and support that I can finally get here.

Chapter 1

Introduction

Android has been the most popular mobile system which occupies over 86.8% market share in Q3 2018 [38]. Along with the popularity, the Android community also faces various threats, such as malware [85], pirated apps [83] and so on. One of the most important mitigation techniques for these threats is the effective and precise dynamic analysis to reveal the underlying sensitive behavior of apps.

Sensitive behavior is normally represented by APIs protected with permissions. Previous work [64, 75] traces sensitive APIs to reconstruct the behavior of Android apps and perform specific analysis, such as malware detection. However, code obfuscation techniques can hinder the analysis based on API tracing. To demonstrate this point, we present an evaluation on the extent to which code-reuse-based techniques can be applied to obfuscate Android apps. Moreover, we extend code-reuse-based obfuscation to the Android platform by proposing an obfuscation mechanism for both Java and native code. Results show that code-reuse-based obfuscation can prevent analyzers to gain a detailed understanding of the obfuscated app and make the API tracing based analysis ineffective.

Although dynamic analysis can use other features, (e.g., system call) to reconstruct sensitive behavior, these techniques require emulators, rooted devices or modified systems to conduct the analysis. Analyzing under these environments requires input generation tools [4, 35, 46, 73] to automatically execute the target apps. This

fact brings in two limitations for existing dynamic analysis techniques. First, input generators cannot provide as wide code coverage as humans. Most input generators can only provide a random series of events, e.g., touching on the screen, to mimic real users' behavior. The random behavior generated by these tools can hardly match the pattern of the real app usage to successfully invoke certain functionalities, e.g., registration. Choudhary et al. compared several popular monkey tools and found that the best statement-level code coverage that these tools can reach is 40% [20]. Bao et al. show that the popular input generators can only reach 30% API-level code coverage. Meanwhile, anti-analysis techniques [39, 66] allow apps to recognize the running environment and hide their sensitive behavior accordingly. For example, apps can detect whether the running environment is emulated based on the GPS info or IMEI number. Moreover, anti-analysis techniques can choose to trigger sensitive behavior only under specific circumstances that have no dependencies on program inputs, e.g., after receiving an SMS, at a particular time slot, or when receiving a remote command [69].

Both the insufficient code coverage of the input generators and the anti-analysis features of the target apps hinder existing dynamic analysis from invoking the potential behavior of them. Theoretically, to enable large-scale deployment and evade anti-analysis techniques, the optimal solution is to conduct the analysis on devices used by the general public. In this thesis, we study to what extent dynamic analysis can be applied to non-rooted and unmodified devices.

Dynamically analyzing apps' behavior on such devices is challenging. Previous tools [10, 14, 24, 63, 64, 77] adopt API tracing, system call tracing and so on, to infer the underlying behavior of the apps. As low-level information (e.g., API call or system call) commonly used by previous work is not accessible on non-rooted devices, these techniques cannot be applied to devices used by the public.

To deal with this problem, we propose a system called UpDroid. Instead of logging low-level events, we monitor the state changing of different types of public resources on the target device. The changes convey information about the sensitive

behavior of the apps. For example, we can monitor message sending behavior by detecting the newly added rows of the content provider `content://sms`. The changing event corresponds to behavior that has been *successfully* performed on the devices, which is different from detecting *attempts* of actions by tracing API calls.

Unlike existing works which can hook into the apps, monitoring the state changes of public resources brings another challenge – identifying the apps that trigger the monitored events. Hence, we use machine learning techniques to build a ranking model for identifying the app from all the running apps at runtime. With both the monitoring component and the app identifying model, UpDroid can monitor various events on non-rooted devices including making phone calls, accessing the camera, reading/writing files and so on.

In order to further investigate to what extent we can analyze Android apps on non-rooted devices, we propose to use side-channel information to infer app behavior with machine learning techniques. The side-channel information we used is the GPU interrupt timing information from the sampling of the `/proc/interrupts` file. Android device has hundreds of interrupts which represent the interaction between hardware/software devices and CPU. Different behavior results in different changes to the interrupts. For example, playing video increases the video decoder interrupt. Hence, the interrupt timing series can be used to feature the application’s behavior. We propose a prototype for inferring app behavior based on the GPU interrupt timing information. We first use a public API to monitor users’ interaction with the device to record when an app is launched, since apps are usually launched by interactions like touching icons on the screen. With interrupt info, we are able to identify which app launches. Moreover, we are also able to identify the app’s behavior when it is launched, e.g., whether the app is used to open a camera or used to browse certain web pages. Experiments show that the activity identification models can effectively disclose app activities on Android. The precision of identifying the launched app can reach around 90%. The precision of identifying activities within an app is higher than 80%. This work shows more possibility of analyzing

via non-rooted devices.

Beyond monitoring app behavior on a single device, we find that dynamic analysis based on non-rooted devices is also effective in analyzing app behavior among multiple devices. It can capture user interactions and features of the running environment which are decisive inputs for inter-device app behavior. We conduct a simulation study for analyzing different factors that would affect the malware spreading with a dynamic monitor on Android devices. Specifically, we choose the spreading behavior of malware and aim to analyze different factors from users and the environment that would affect the spreading among multiple devices. We build a system to trace malware spreading messages among users' devices and conduct the simulation study with the system. The experiments present the possibility of deploying dynamic analysis on real users' device for analyzing malware spreading. The results shows that spreading method, relationship and contact frequency would significantly affect the spreading of the malware.

Chapter 2

Literature Review

We describe related work of our proposed techniques respectively. We first introduce obfuscation techniques for hiding app behavior. Then, we describe existing work of dynamic analysis on Android. Finally, we introduce side-channel techniques which give insights to dynamic analysis on non-rooted devices.

2.1 Code Obfuscation Techniques

Traditionally, there have been three categories of obfuscation techniques proposed, including layout obfuscation [17], control-flow transformation [22, 70], and data obfuscation. Layout obfuscation [17] removes relevant information from the code without changing its behavior. Control-flow transformation [22, 70] alters the original flow of the application. Data obfuscation obfuscates data and data structures in the application. These techniques are certainly helpful in obfuscating Android apps; however, they are not specific to the Android platform in the implementation aspect and are not effective in hiding app behaviors that implemented by sensitive APIs. The code-resuse based technique we propose is specific for Android platform and can effectively hide sensitive APIs.

There are also free or commercial obfuscation techniques specifically provided to Android developers. ProGuard [42] is a free and commonly used one that ob-

fuscates the names of classes, fields, and methods. DexGuard [1] is a commercial optimizer and obfuscator. It provides advanced obfuscation techniques for Android development, including control-flow obfuscation, class encryption, and so on. DexProtector [2] is another commercial obfuscator that provides code obfuscation as well as resource obfuscation, such as the Android manifest file. Current obfuscation techniques for Android mainly target at Java code and cannot hide the sensitive APIs used for conducting privacy or security-related behavior. The proposed obfuscation technique in this thesis can significantly hide the sensitive APIs in both Java and native code.

Code reuse techniques, including Return-into-lib(c) [52, 65], Return-oriented programming [13, 61] and Jump-oriented programming [11, 18, 25], are first proposed to exploit vulnerable apps by hijacking their control-flow transfers and constructing malicious code dynamically. Among these code-reuse techniques, only a few of them work on Android system or the ARM architecture. Davi et. al. [25] proposes a systematic jump-oriented programming technique on the ARM architecture. The gadget set proposed in this work consists of gadgets ending with BLX instructions. In this work, we propose to use a different type of gadgets that are more commonly found in native libraries and apply this technique to code hiding on Android platform.

Recently, several code-reuse-based obfuscation techniques [44,45,68] have been proposed. One of the code-reuse-based obfuscation techniques is RopSteg — a steganography technique on x86 [44]. RopSteg protects binary code on x86 architecture, while our code-reuse-based obfuscation on Android platform works for both Java and native code on Android platform. Another work [68] proposes a malware named Jekyll which hides malicious code and reconstructs it at runtime. In this work, we propose a complete system with semi-automatic tools for code obfuscation on Android. Our obfuscation mechanism can be used for protection of either malicious or benign code.

2.2 Dynamic Analysis on Android

Various dynamic tools/platforms have been proposed for analyzing underlying behavior of Android apps. DroidScope [77], CopperDroid [64], VetDroid [82], DroidBox [43] and other tools analyze the API calls, system calls, or other features to reconstruct app behavior. For example, CopperDroid can reconstruct the apps' behavior, e.g., sending SMS, by observing and dissecting the system calls. These tools are based on app instrumentation, framework modification or emulator instrumentation. Hence, they are not applicable to the devices used by the general public and need input generator tools to automatically run the target apps. Our work can also detect sensitive behavior of Android apps. The advantage is that our work can be applied to devices used by the general public while these tools have critical requirements for either the running environment or the target apps.

Andromaly [60], CrowdDroid [14] and other tools can also run on non-rooted devices for app analyzing. These tools detect runtime features, e.g., system call logs and side channel info, of the running apps and use these features to identify whether the app is benign or not with machine learning techniques. Our work gathers the running features and uses machine learning techniques to identify the apps that invoke the captured events. Meanwhile, our work generates fine-grained reports of apps, while these tools only classify the apps. App Guardian [81] also gathers side channel info and detects malicious behavior, e.g., on non-rooted devices. However, the detection is based on specific heuristics and only targets runtime information gathering attacks. Compared to this paper, the analyzing techniques we propose is generic for app activities, such as sensitive behavior, app launching and activities within an app.

BareDroid [51], Ninja [53], Njas [9] and other works provide dynamic analysis techniques which are resistant to anti-analysis techniques. BareDroid is an analysis system which uses a phone cloud for the analysis. It needs to customize the devices in the phone cloud and thus cannot be applied to devices used by the public. Ninja

needs to customize the firmware on the Android devices. It is also difficult to be applied to devices used by the public. Njas provides sandboxing for unmodified apps on non-rooted devices. It dynamically loads the target app's APK file to the sandboxing app's context for fully accessing the target app's resources and runtime state. Njas relies on an app database and needs to obtain the APK file of the target app at the same version. Njas cannot sandbox the apps which do not have readable APK files, e.g., the paid apps. Although these analysis systems are transparent to anti-analysis techniques, they cannot be applied to devices used by the general public directly. Compared to these systems, UpDroid can be easily deployed on the users' devices without any modification to the systems or any requirement on the target apps.

2.3 Side channel and crowd sourcing

As well as current dynamic analysis techniques, other works also give us inspiration about the runtime monitoring. The information provided by proc file system is an important resources for app analyzing. Zhang and Wang [80] inferred keystrokes by disclosing ESP with `/proc/<pid>/stat`. Qian et. al. [55] disclosed TCP sequence numbers from information provide by the `/proc/net/` directory. Zhou et.al. [84] disclosed users' private information, such as user identities and location, by analyzing procfs on Android. Diao et al. [28] propose to use the interrupt time series produced by the touchscreen controller to infer the unlock pattern and foreground app. In this paper, they proposed to use series of the total amount of interrupt as the feature to infer activities. In this thesis, we specifically choose GPU interrupt timing information obtained from `/proc/interrupts` to analyze app activities on the device without obtaining root permission or modifying the system. In our work, we specifically choose interrupts generated by GPU, which is the essential hardware for generating user interfaces in modern systems. MopEye [74] leverages the `VpnService` API to monitor the network usage of the apps. UpDroid also

uses this technique to detect network activities on the devices.

Analysis of non-rooted devices requires crowdsourcing in deployment. Crowdsourcing has been used by different techniques for enhancing mobile security. DroidNet [56] provides a framework for users to install applications and use crowdsourcing to search for expert users for permission control. Droidganger [79] allows collections of users of the same application to help each other on permission understanding by sharing their permission reviews. CrowdDroid [14] is a typical work that uses crowdsourcing for malware analysis. It uses 20 clients which are real running devices to run their application and gathering features of applications for analysis. In this work, the crowdsourcing method we used is similar to crowdsourcing, we recruit participants to run applications as the data collector and use the data for the analysis.

Chapter 3

Obfuscation of Sensitive Behavior

The goal of this chapter is to demonstrate that sensitive behavior analysis based on tracing API calls can be hindered by obfuscation techniques. This chapter investigates the code resources for code-reuse technique in Android apps and presents that the code-reuse-based obfuscation is effective for hiding sensitive API calls.

3.1 Introduction

Sensitive API analysis is widely used by researchers to reveal app's underlying behavior. Obfuscation is one of the methods which would make the analysis more difficult. Traditional Java obfuscation techniques [17, 21, 22] only apply relatively simple obfuscation schemes, e.g., renaming identifiers and removing debugging information. Although identifiers of classes and methods are no longer understandable after the obfuscation, names of the system APIs and the control flow of the program still enable reverse engineering to a great extent. Hence, it is still easy to analyze the sensitive system APIs by either static or dynamic analysis. To hinder the analysis, one of the possible ways is to obfuscate sensitive APIs by code-reuse techniques.

Return-Oriented Programming (ROP), which belongs to the bigger family of code-reuse-based techniques, was recently proposed as an attacking technique to

exploit vulnerable programs [11, 13, 18, 25, 61]. It was subsequently used for code protection [44, 45, 68] and to provide program steganography, e.g., RopSteg [44]. The main idea of code-reuse-based obfuscation is to replace essential code with small code pieces distributed in the app and to reconstruct the essential code dynamically. These small code pieces, typically ending with return/return-like instructions, are called gadgets. Then, a payload, which contains addresses of the gadgets and parameters needed by them, is generated for code reusing. This payload is typically used to trigger some vulnerability (e.g., buffer overflow) and to invoke the hidden code by executing the selected gadgets one by one. With this technique, the semantics of the essential code in the original program are hidden in the payload. As part of the data in an app, payload is safer than the original code under the disclosure of reverse engineering tools. The hidden code can be further protected through dynamically downloading the payload from a trusted remote server. In addition to protecting benign code, this technique can also be used for hiding malicious behaviors by adversaries.

However, RopSteg and other code-reuse-based techniques cannot be directly applied to Android applications. First, Android apps are mainly developed in Java, while code-reuse-based techniques are based on native binaries typically compiled from C/C++. Second, Android devices are built on ARM architecture on which registers are used for parsing function parameters and saving return addresses [59], as opposed to x86 which is more dependent on the stack.

In this chapter, we present the first evaluation on the extent to which code-reuse-based techniques can be applied on Android application obfuscation. Moreover, we propose an effective code-reuse-based obfuscation mechanism for Android apps. This mechanism helps developers to obfuscate small pieces of sensitive code, including both Java and native code. We evaluate gadgets found in binaries of Android apps and calculate the amount of gadgets in several common native libraries used by Android apps. Results show that 835 gadgets in the C standard library (`libc.so`) cover a Turing complete gadget set. We implement this idea in a tool

called AndroidCubo (Android Code-reuse Based Obfuscation) and successfully apply it on real examples to protect both Java and native code with a small overhead. We show that the security of our obfuscated code is comparable to that obfuscated with Java reflection.

3.2 Overview

Android app obfuscation focuses on preventing reverse engineering by adversaries. We assume a threat model in which an adversary reveals essential code in Android apps with reverse engineering tools, such as Apktool and APKstudio. These tools help adversaries decompile Android APK and disassemble the resources to Java or assembly code. Then, adversaries can tamper the decompiled app and repackage it to perform malicious behaviors. Obviously, we assume that source code of the Android app is not available to the adversary.

An effective obfuscation technique has to achieve two goals when targeting Android applications. First, it should protect the compiled essential code from being reverse engineered to a human understandable format. Second, it should be generally applicable to any code segments to be hidden on any Android applications. In the context of code-reuse-based techniques, this means that a Turing complete gadget set that consists of frequently appeared gadgets is needed.

Fig 3.1 gives an overview of our code-reuse-based technique in obfuscating the essential code in an Android app. First, the essential code is replaced with a gadget sequence based on the Turing complete gadget set. The gadget sequence represents the semantics of the essential code and is also regarded as the code reuse program. Next, we prepare a payload according to the gadget sequence. After that, a segment of trigger code is embedded in the app to invoke the protected code at runtime. At last, when the protected app is running, the payload will be loaded into the memory of the app and passed to the trigger code for invoking the protected code.

The Turing complete gadget set is a fundamental requirement in this technique

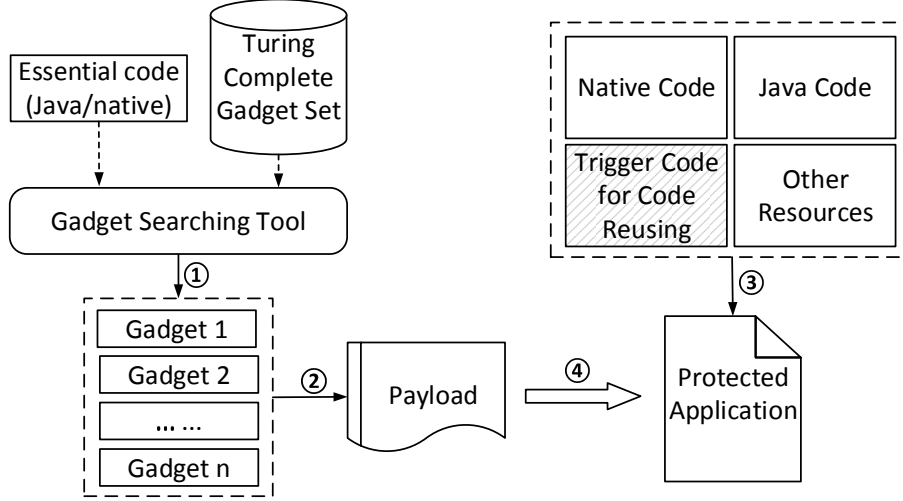


Figure 3.1: An overview of our code-reuse-based obfuscation technique for Android apps.

for providing enough gadgets to substitute the essential code. In the following sections, we first present our analysis of gadgets on ARM and then discuss the details of the code obfuscation mechanism.

3.3 Turing Complete Gadget Set

As we discuss in the earlier section, having a Turing complete gadget set is a necessary condition for a code-reuse-based obfuscation technique to be generally applicable to most Android applications. In this section, we present a Turing complete gadget set found available for code reuse obfuscation on ARM architecture. We also analyze the number of available gadgets in each category. We focus our analysis on Android 4.4 on a Nexus 5 handset. In the following description, R_a-R_d and R_x-R_y denote different registers of ARM.

Previous studies [25, 26] applied gadgets ending with `BLX R_a` in their code-reuse techniques. `BLX R_a` is an indirect jump instruction whose jump destination is specified by register R_a . Unlike return instructions, `BLX` cannot fetch gadget addresses from memory. Thus, a specific kind of gadget, called update-load-branch (ULB) gadget, is used to sequentially fetch gadget addresses to registers and chain

the gadgets together. However, the ULB gadget is very hard to find in native libraries [25]. Besides that, this strategy doubles the length of the gadget sequence, which makes code-reuse-based obfuscation techniques more complicated and slows down the program. Hence, we explore the possibility in using another type of gadgets that ends with `POP {Rx-Ry, PC}`. This `POP` instruction loads an address from the stack to the program counter register `PC` directly. It always appears in the epilogue of a function and is more commonly found in native libraries than the `BLX` instruction.

Our gadget searching strategy is to look for basic blocks (instruction sequences that do not contain branches) ending with a `POP {Rx-Ry, PC}` instruction to minimize the effort needed to handle branches in instruction sequences and payload generation. We implement this strategy into a gadget searching tool in python. This tool searches for all available gadgets and their relative addresses in native libraries. It also categorizes the available gadgets to different classes according to their functionality.

We apply our gadget searching tool on several commonly used native libraries used by Android apps and compare the number of available gadgets in our gadgets set with that in the gadget set proposed by Davi et al. [25], see Table 3.1. The results show that number of gadgets in our gadget set is much larger than that used by Davi et al. [25]. This is because `POP {Rx-Ry, PC}` is more frequently used than `BLX Ra` in the native libraries. With the larger number of gadgets, the probability of finding all gadgets needed in the Turing complete gadget set is higher. Besides that, the more gadgets we find, the more flexibility we have for essential code replacement.

Table 3.1: Number of gadgets found in different gadget sets.

Native Libraries	libc	libruntime	libunity	libvideo	libcocos2d
# of Gadgets (Our Gadget Set)	835	2,244	21,483	317	12,913
# of Gadgets (Gadget Set in [25])	77	1,326	10,734	148	6,126

Upon our analysis, we realized that gadgets that implement basic operations, such as memory operations, arithmetic, and logic operations, can be easily found through searching the corresponding instructions. Other functionality, including control-flow transfers and function calls, need to be constructed more carefully. We carefully analyzed the gadget sets found and managed to form a Turing complete gadget set for converting sensitive code into gadget sequences, see Table 3.2.

Table 3.2: Number of different types of gadgets in our gadget set.

Gadget Functionality	libc	libruntime	libunity	libvideo	libcocos2d
Load	127	151	2,484	60	1,607
Store	227	161	5,518	77	2,333
Add	20	3	878	23	204
Sub	30	1	78	3	35
Shift	12	8	20	2	689
And	6	8	137	3	60
Or	21	6	274	3	100
Xor	2	2	31	0	22
Unconditional Branch	226	753	12,063	84	3,035
Conditional Branch	28	15	1,107	29	29
Function Call	8	187	865	5	458

The results show that libraries contain sufficient gadgets in each category of the Turing complete gadget set, with the exception of `libvideo` where there is no gadgets to perform `xor` operation. However, also note that `xor` could be indirectly implemented with other logical operators. This shows that many commonly used libraries are sufficient for providing gadgets for code-reuse obfuscation.

3.4 Code Obfuscation

With the Turing complete gadget set found in various native libraries covering different functionality, we now present details of the obfuscation mechanism for protecting a piece of essential code in an Android application. The code protection process, as shown in Figure 4.1, consists of a few steps in 1) replacing the sensitive code with our gadget sequence; 2) generating code-reuse payload according to the gadget sequence; and 3) constructing trigger code to invoke the hidden code with payload in the app.

3.4.1 Essential Code Replacement

It is usually straightforward to replace the essential code to be obfuscated with gadget sequences. Most code-reuse techniques typically disassemble the essential code to instruction sequences first, and then substitute them with semantically equivalent gadgets. However, dealing with Android applications makes this process more complicated as we want to be able to obfuscate both the native and Java code. This makes our code-reuse-based obfuscation tool different from most existing ones.

For Android apps, native code is always compiled to native libraries (`.so` file) by the building module of Android Native Development Kit (NDK). Reverse engineering tools, such as IDAPro, Hopper, or the GNU Project debugger (GDB) can be used to disassemble the native libraries and to obtain the instruction sequences for the essential code to be obfuscated. We can then substitute instructions in the essential code with gadgets in the native binaries of the app. Since most of these native libraries contain Turing Complete gadget sets as shown in Table 3.2, we will always be able to perform this substitution successfully.

Dealing with Java code in Android apps is more challenging, since existing code-reuse techniques only support native code. Although a subset of the language-independent functionality (e.g., concatenation of strings can be implemented in Java as `+` operator and native code as `strcat()` method) can be implemented in native code as well, other functionality that uses classes or methods specifically provided by Java or Android cannot be directly implemented in native code (e.g., enable bluetooth can only be implemented in Java as `BluetoothAdapter.enable()`).

Fortunately, the Java Native Interface (JNI) provides a flexible connection for the communication between Java and native code [32]. JNI provides several native methods for accessing object's field from native code as well as methods for converting Java classes to native classes, including `GetObjectClass()`, `GetMethodID()` and `CallVoidMethod()`. These methods allow native code to use Java class objects and to call Java methods by providing corresponding class

names and method names. In addition, JNI also provides methods to convert Java objects to native variables. For example, `GetStringUTFChars()` can be used to convert a Java string to native chars.

Fig 3.2 shows an example of the corresponding native code that can be used to replace a sensitive Java API `sendTextMessage()`. In this example, The JNI function `CallVoidMethod()` will call the sensitive API in native code after retrieving the class and method names.

```

1 void * sendSMS(JNIEnv *env)
2 {
3     jclass smsclass = env->FindClass("android/telephony/SmsManager");
4     jmethodID get = env->GetStaticMethodID(smsclass, "getDefault", "()
        Landroid/telephony/SmsManager;");
5     jobject sms = env->NewObject(smsclass, get);
6     //Obtaining sendTextMessage()
7     jmethodID sendMethod = env->GetMethodID(smsclass, "sendTextMessage",
8         "(Ljava/lang/String;Ljava/lang/String;Ljava/lang/String;Landroid/app/
        PendingIntent;Landroid/app/PendingIntent;)V");
9     jstring destAddress = env->NewStringUTF("1234567890"); //Phone number
10    jstring text = env->NewStringUTF("native"); //SMS content
11
12    //Sending SMS with sendTextMessage() in native code
13    env->CallVoidMethod(sms, sendMethod, destAddress, NULL, text, NULL, NULL)
14    ;
15 }

```

Figure 3.2: The native code of calling `sendTextMessage()` with JNI.

In addition to the proposed method of implementing Java functionality in native code via JNI and then subsequently obfuscating the resulting native code, here we propose another method using shell command. We notice that many Java operations can be represented with shell commands in Android apps, e.g., reading SMS can be implemented through shell command `content query --uri content://sms`. Therefore, we propose to obfuscate Java code by first replacing it with a call to `system()` with the corresponding shell command, and then subsequently obfuscating the calling of `system()` with our code-reuse program. This method only needs two gadgets — the first one to move the address of the corresponding command to register `R0`, and the second to invoke the system call function. The actual shell command appears as parameters to the system call.

Table 3.3 presents some common behaviors which can be represented by shell

commands on Android. These commands are all feasible to be used on normal Android devices. The available shell commands can be found under the directory `/system/bin` in the corresponding Android devices. More complicated operations can be hidden in shell scripts written with available commands and be invoked through executing the scripts with `system()`. These shell commands include simple ones like file operations, process management, network configuration, as well as those provided by Android Debug Bridge (ADB) for activity management and package management.

Table 3.3: Examples of operations on Android and the corresponding shell commands.

Operations	Shell Command
Open Messenger	<code>am start --user 0 -a android.intent.action.SENDTO -d sms:PHONE_NUMBER --es sms_body MESSAGE</code>
Read SMS	<code>content query --uri content://sms</code>
Open Dialer	<code>am start --user 0 -a android.intent.action.DIAL -d tel:PHONE_NUMBER</code>
Start Browser	<code>am start --user 0 -a android.intent.action.VIEW -d URL</code>
Create Directory	<code>mkdir DIRECTORY_PATH</code>

3.4.2 Payload Generation

The main advantage of code-reuse-based obfuscation tools over other obfuscation techniques is that the hidden code exists in the form of data rather than instructions. To achieve this, we need to prepare a payload according to the gadget sequence. Payload is a segment of memory content that contains semantics of the protected code and will be used for overwriting control data at runtime. A payload typically consists of three parts. The first part is the data that will be used to overwrite control data in memory to redirect control flow to the hidden code. The second part consists of the parameters and addresses for the gadget sequence which presents the semantics of the hidden program. The third part is a segment of buffer with data

needed by the code reuse program and other padding data. Fig 3.3 is an example of the payload which has been loaded on the stack.

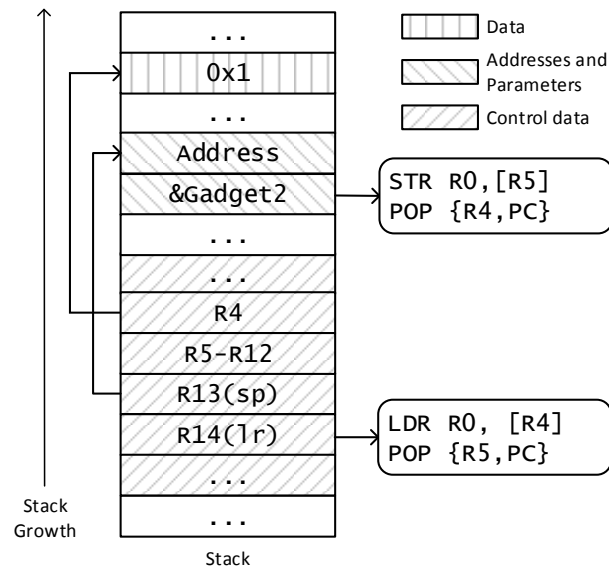


Figure 3.3: Layout of the payload. The shadowed areas present different parts of the payload.

This payload is used for the gadget sequence that loads a number `0x1` from memory and stores it at another address. From bottom to top of the stack, the first part is the data that overwrites control data `jmp_buf` which is used to set register values of the execution environment. In the rewritten `jmp_buf`, `R4` is set to the address of `0x1` and stack pointer is set to the beginning of the second part of the payload. The second part contains the parameter needed by the first gadget and the address of the second gadget. The last part of the payload contains other data — the number `0x1` to be loaded from memory and stored to the address specified by `R5`. To generate the payload, the most essential steps are to store the address of the first gadget in `lr` and addresses of following gadgets on the stack. Thus, by changing `sp`, the gadgets will be executed in proper order.

3.4.3 Code Triggering

After preparing the payload, extra code needs to be added to the app as an entry point of the hidden code. This part of the code fetches the payload at runtime and uses

it to trigger the code-reuse program. Code-reuse programs are commonly triggered through overwriting control data, including return addresses, function pointers, and jump buffer. The overwriting could be based on a set of vulnerable library functions that lack boundary checking, such as `gets()`, `fread()`, `strcpy()`, and `sprintf()`. As in some existing work [25], the control data we choose to overwrite is the `jmp_buf` structure that is used to restore the execution environment in exception handling. The `jmp_buf` structure contains data that will be used to set values of registers which are used for storing parameters and the return address of a function call. Thus, it is convenient to redirect the control flow through overwriting `jmp_buf` structure on ARM.

Fig 3.4 shows an example of overwriting `jmp_buf` [25]. In this piece of code, function `setjmp()` and `longjmp()` are used to store and restore the execution context in variable `jbuf`. Reading data from `sFile` to `buf` will overwrite `jbuf`. Thus, `longjmp()` will direct the program execution to somewhere specified by the overwritten `jbuf`.

```
1 typedef struct foo{
2     char buf[JP_BUFSIZE];
3     jmp_buf jbuf;
4 }FOO, *PFOO;
5 PFOO f;
6
7 void * overflow(char * filePath)
8 {
9     int i;
10    ... ..
11    i = setjmp(f->jbuf);
12    fread(f->buf, 1, BUFSIZE+256, sFile);
13    ... ..
14    longjmp(f->jbuf, 2);
15    ... ..
16 }
```

Figure 3.4: Trigger code to be added to source code of the application.

3.4.4 Payload Protection

Since the semantics of the essential code are hidden in the code-reuse payload, it is important that our obfuscation tool provides protection on the payload to resist

reverse engineering attempts. To protect the payload, we propose three possible solutions.

- Instead of storing payload as static resources of the Android app, the payload can be embedded in the resources using information hiding techniques. For example, the payload can be hidden in a segment of normal code, e.g., as an image, using steganography techniques [50].
- The payload can exist in an encrypted form of data in the Android app, and be decrypted at runtime.
- To completely remove the payload from the APK file of the Android app, we can dynamically download it from a trusted remote server [68]. Dynamically, the app will request and receive payload from the server based on a reliable protocol.

In this work, we use the last, and the most secure, method.

3.5 Implementation and Case Studies

We manage to implemente our idea of obfuscating Android application as a tool set, AndroidCubo. AndroidCubo takes as input the source code of an Android app and obfuscates selected native and Java code in it. We present some implementation details and applications of AndroidCubo on an app in this section. Experiments were performed on a Nexus 5 running Android 4.4.

3.5.1 Implementation details

Code-reuse programming is complicated since it involves a lot of low level operations on memory and registers. We implement AndroidCubo as a tool set for helping Android app developers to obfuscate sensitive code with our code-reuse technique.

It contains a source code template to be inserted into the Android source code and a payload maintainer to execute on a trusted server.

The source code template contains a Java class named `ObfuscateUtil` and a C program named `Hiding`. The class `ObfuscateUtil` provides native interfaces for calling native methods in `Hiding`. It also implements network communication with the trusted server which maintains the payload for the code-reuse program. The `Hiding` program has a method named `trigger()` that uses the payload (received from communication with the trusted server) to trigger the obfuscated code.

This source code template can be directly added to the Android project for obfuscating a segment of sensitive code. The only additional code a developer has to add is for preparing parameters if they are obfuscating API calls. To use this template for obfuscating multiple segments of sensitive code, the user needs to add trigger methods in `Hiding` and the corresponding interfaces in `ObfuscateUtil`.

The payload maintainer on the server side has two parts. The first part is a payload generator that works in the following manner.

- **Native code obfuscation** Our gadget searching tool lists available gadgets and their relative addresses for the developer to construct the gadget sequence. The developer can also use other existing tools, e.g. ROPgadget [57] or Q [58], to develop their code reuse program.
- **Java code obfuscation through shell commands** The generator automatically generates the payload with a command provided by the user.
- **Java API obfuscation** The developer specifies the addresses of the API and the corresponding parameters and our generator outputs the payload.

The second part is a program for sending payload to the app. This program is developed with PHP with which the server will handle the request of payload from the app, trigger the payload generator, and then send the payload over to the app.

3.5.2 Case study: Obfuscating Native Code

To demonstrate AndroidCubo in obfuscating native code, we hide a simple comparison algorithm as shown Fig 3.5(a)(b). This algorithm obtains and stores the larger one of the two input numbers. As described in Section 3.4, this simple algorithm needs to be converted to a sequence of gadgets first. AndroidCubo first executes the gadget searching tool and finds available gadgets and their relative addresses, and then generates a sequence of gadgets to substitute the original code as shown in Fig 3.5(c). In this sequence, gadgets 1-3 are used to load the first operand to register R9. Gadgets 4-6 are used to load the second operand to register R3. The last conditional gadget is used to find and store the larger number.

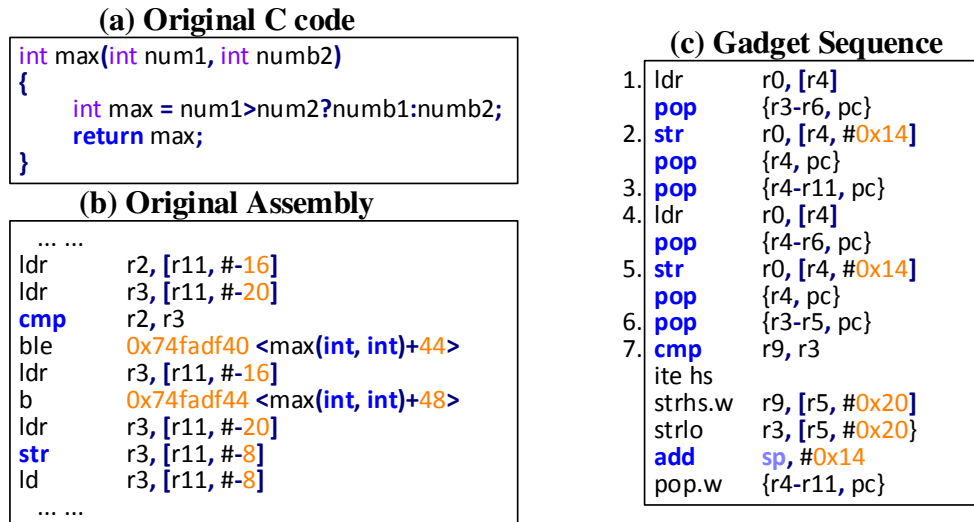


Figure 3.5: Source code to be hidden and the corresponding gadget sequence. (a) Original C code; (b) Original assembly code; (c) Gadget sequence.

AndroidCubo then generates the payload based on the gadget sequence. In particular, the first part of the payload is the data used to overwrite the control data `jmp_buf`. `jmp_buf` directs the stack pointer to the beginning of the second part — the addresses and parameters of the gadgets. `LR` is then set to the address of the first gadget. The last part of the payload is a buffer containing junk data.

We recompile the Android app with outputs from AndroidCubo and execute the app with the corresponding payload. After executing the app and loading the payload to the stack, `longjmp()` successfully executes with the prepared `jmp_buf`,

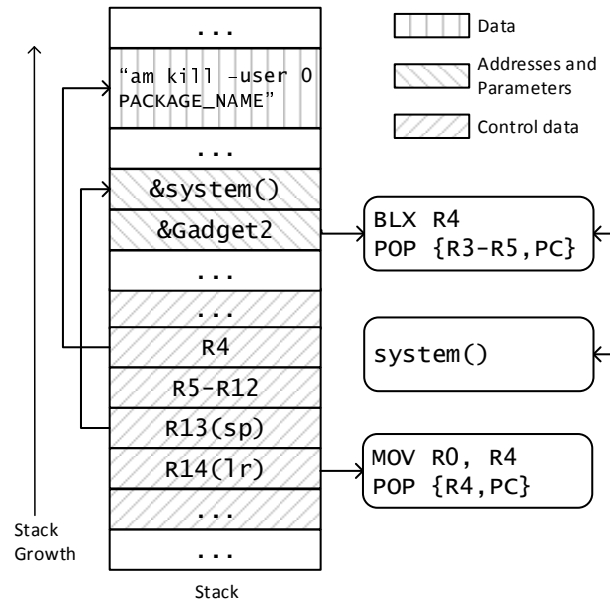


Figure 3.6: Stack layout after loading the payload.

and the gadget pointed to by LR executes followed by other gadgets prepared in the payload.

3.5.3 Case study: Obfuscating Java Code

We use another example to demonstrate using AndroidCubo to obfuscate Java code. In this example, we hide the Java code that kills a background process.

The operation of killing a background process is typically implemented by obtaining an `ActivityManager` object and killing the process by calling the method `killBackgroundProcess()` in Java. AndroidCubo hides this Java code through a shell command `am kill --user 0 PACKAGE_NAME` with two gadgets. The first gadget `MOV R0, R4; POP {R4, PC}` is used to prepare the shell command as a parameter for `system()`. The second gadget is a function call gadget `BLX R4; POP {R3-R5, PC}` to invoke the shell command. Fig 3.6 presents a view of the stack after our app loads the payload generated by AndroidCubo to overwrite a buffer.

From bottom to top of the stack, the three shadowed areas present the corresponding parts of the payload. The first part is the overwriting of control data

`jmp_buf`. In `jmp_buf`, register `LR` is set to the address of the first gadget. Function pointer `SP` is set to the beginning of the second part of the payload. Register `R4` is set to the address of the command that will be assigned to `R0` as the parameter of `system()`. The second part is the gadget addresses and parameters. The most essential data on this part is the address of `system()` and the address of the second gadget. The last part includes the padding data and the command string needed by `system()`.

3.5.4 Overhead

In our experiments in applying `AndroidCubo` to the Android apps, it introduces around 150 LOC to native part and around 250 LOC to Java part of the Android application.

3.6 Comparison with other Obfuscation Techniques

There have been existing obfuscation techniques proposed, and in this section, we conduct a comparative test on sensitive API obfuscation among our code-reuse-based method and other techniques, including control-flow obfuscation and Java-reflection-based obfuscation. Control-flow obfuscation techniques typically hide or protect the selected code by branching or looping garbage code. Java-reflection-based techniques typically hide sensitive API calls by using Java reflection to access the APIs through their names. We use these techniques to obfuscate an open source application named `OverFlow`. The sensitive API that we target to obfuscate is `sendMessage()`.

3.6.1 The Experiment

We obfuscate the target app with all three techniques and then build the signed APK file. We use `Apktool` [72], `dex2jar` [3], and `JD-GUI` [29] to reverse engineer

the APK files obtained to see how much information of the sensitive API can be reconstructed. Apktool is used to unpack the APK file and obtain the dex file. dex2jar converts the dex file to jar files which contain the byte code of the app. After obtaining the jar file, we extract the class files in the jar and use JD-GUI to reverse engineer class files to readable Java code. The above constitutes the most commonly used methods for reverse engineering Android apps.

3.6.2 Reverse Engineering Results

Fig. 3.7 presents the reverse engineering output for the un-obfuscated app (Fig. 3.7(a)) and apps obfuscated by the three different techniques (Fig. 3.7(b)-(d)).

Although the control flow recovered in Fig. 3.7(b) seems opaque, it is easy to spot out the sensitive API call from the byte code at line 9. This shows that the control-flow obfuscation manages to introduce confusion in terms of how control transfers, but it fails to hide the existence of Java API call. From Fig. 3.7(c), we can also easily figure out the name of the API from the first parameter of `getMethod()`.

Fig. 3.7(d), on the other hand, substitutes the sensitive API call with a native function call whose functionality cannot be inferred from the name. That said, one could further analyze the native function `CallVoidMethod()` to see if it contains any hints of the API function to be called. We use IDAPro to reverse engineer the native function `CallVoidMethod()`, and find that the string `sendMessage` and `(Ljava/lang/String;...)` can be recovered from the binaries.

3.6.3 Discussion

In our experiments of obfuscating the Android app with different obfuscation methods, AndroidCubo presents better security in hiding the sensitive API call from reverse engineering tools. At a high level, its idea is similar to Java-Reflection-based techniques in that both techniques replace the original Java call with another


```

1 private void sendMessage(String paramString1, String paramString2)
2 {
3     try
4     {
5         SmsManager.getDefault().sendTextMessage(paramString1, null,
6             paramString2, null, null);
7         return;
8     }
9     catch (Exception paramString1) { ... }

```

(a) Decompiled code of un-obfuscated sendTextMessage()

```

1 ... ..
2 131: goto -7 -> 124
3 134: aload 4
4 136: aload_1
5 137: aconst_null
6 138: aload_2
7 139: aconst_null
8 140: aconst_null
9 141: invokevirtual 105 android/telephony/SmsManager:sendTextMessage
10 ...
11 144: return
12 145: astore_1
13 146: aload 5
14 148: astore_2
15 ... ..

```

(b) Decompiled code of function call obfuscated by control-flow obfuscation

```

1 private void sendMessage(String paramString1, String paramString2)
2 {
3     try
4     {
5         SmsManager localSmsManager = SmsManager.getDefault();
6         localSmsManager.getClass().getMethod("sendTextMessage", new
7             Class[] { String.class, String.class, String.class,
8             PendingIntent.class, PendingIntent.class }).invoke(
9             localSmsManager, new Object[] { paramString1, null, paramString2
10                , null, null });
11         return;
12     }
13     catch (Exception paramString1) { ... }
14 }

```

(c) Decompiled code of function call obfuscated by Java Reflection

```

1 private void sendMessage(String paramString1, String paramString2)
2 {
3     try
4     {
5         nativeMethod(paramString1, null, paramString2, null, null);
6         return;
7     }
8     catch (Exception paramString1) { ... }
9 }

```

(d) Decompiled code of function call obfuscated by AndroidCubo

Figure 3.7: The decompiled code of calling sendTextMessage() and the decompiled code from obfuscated calling.

method call, and both techniques specify the underlying method to be called via a string. However, the replacement in Java-Reflection-based techniques is still a Java method call, which is relatively easy to analyze; on the other hand, AndroidCubo uses a replacement of native calls that are more difficult to analyze. Coupled with other string obfuscation techniques, we argue that AndroidCubo presents higher resilience in obfuscation compared to Java-Reflection-based techniques.

3.6.4 Limitations

Although applying our code-reuse-based obfuscation technique is feasible, there are a couple of limitations that are worth noting. First, AndroidCubo, in its current form, is a semi-automatic tool. Piecing together gadgets and writing long code-reuse programs are still a complicated process that requires the developer's attention and help. Second, applying code-reuse techniques for good, e.g., in obfuscating program logic, runs into the risk of being prohibited by code-reuse protection mechanisms. That side, current Android systems have no protection mechanisms to resist code-reuse programs, and many advanced techniques [16, 27, 31, 62] are powerful enough to bypass most protection mechanisms.

3.7 Summary

In this chapter, we present a code-reuse-based technique for protecting Android applications. This technique enhances the concealment of both Java and native code in Android apps through hiding essential code. Our evaluation shows that the limited binary resources in Android apps are sufficient for applying code-reuse-based obfuscations. We further implement AndroidCubo semi-automate the process of obfuscating essential code. Examples present that it is practical to protect applications with AndroidCubo. This work is done and published on ICISC 2016. Since obfuscation is effective in bypassing the analysis of code-level features, the next chapters present our efforts on analysis of apps without code-level features.

Chapter 4

Sensitive Behavior Analysis

This chapter demonstrates the feasibility of dynamic analysis on non-rooted Android devices used by the public which does not rely on code level features that can be concealed by obfuscation techniques. It proposes a system called UpDroid for analyzing sensitive app behavior on non-rooted devices. The results show that UpDroid can detect various sensitive behavior with small runtime overhead.

4.1 Introduction

Existing dynamic analysis based on tracing low-level information (e.g., API calls or system calls) requires emulators or modified systems. Hence, these techniques need input generators to conduct the analysis. However, input generators cannot provide as wide code coverage as humans. Previous research shows the best statement-level code coverage that the popular input generators can reach is 40% [20], and best API-level code coverage is 30%. Meanwhile, anti-analysis techniques [39,66] allow apps to recognize the running environment and hide their sensitive behavior accordingly. In this chapter, we propose to dynamically analyze Android applications on non-rooted devices used by the public to gain better code coverage and real running environment.

In this chapter, we propose a system called UpDroid which works on non-rooted

devices without system modification. Instead of logging low-level events, we monitor the state changing of different types of public resources on the target device. The changes convey information about the sensitive behavior of the apps. For example, we can monitor message sending behavior by detecting the newly added rows of the content provider `content://sms`. The changing event corresponds to behavior that has been *successfully* performed on the devices, which is different from detecting *attempts* of actions through tracing API calls. Since Android provides the public APIs for third-party applications to monitor the resources, UpDroid does not require rooting the device or modifying the system.

Unlike existing works which can hook into the apps, monitoring the state changes of public resources brings another challenge – identifying the apps that trigger the monitored events. Hence, we use machine learning techniques to build a model for identifying the apps at runtime.

UpDroid can monitor various events including making phone calls, accessing the camera, reading/writing files and so on. It achieves around 80% precision in identifying the apps that trigger the observed events. We compare UpDroid with the traditional API hooking methods to study how far UpDroid can go in covering different types of behavior and how it is different from the traditional hooking method. Experimental results demonstrate that the events UpDroid can capture cover 15 out of 26 dangerous permissions, while API hooking covers 21. The permissions covered by UpDroid contain the popular ones used by both malware and benign apps. From tests on several popular apps, we observe that UpDroid detects the result-based events and API hooking misses some because of the incompleteness of the sensitive API list.

4.2 Background and Motivation

In this section, we introduce some background information for this work and present the current state of sensitive behavior monitoring on Android to motivate this work.

4.2.1 Resources and Observers

Android has mature security protection mechanisms based on its permission model and the security features inherited from the Linux kernel. Guarded by these mechanisms, third-party apps have limited access to the static and runtime resources of the device. Normally, only with legal permission declaring and requesting an app could access the protected resources and perform sensitive behavior.

We propose to monitor state changing of four categories of resources which are normally available for third-party apps to detect the sensitive behavior.

Content Provider Content provider is an app component provided by Android for managing access to a structured set of data. It is often used to store users' personal information, such as SMS, call logs, contact information and so on [33]. Content provider encapsulates the data and provides mechanisms for security. With proper permission, third-party apps can access the content providers that are open to external apps. Various functionalities are implemented with content providers, e.g., the default app for sending and receiving SMS uses content provider to store the SMS logs. Android provides the `ContentObserver` API for receiving callbacks of changes to a content provider to monitor the content provider events. For example, a malware named HongTouTou uses this API to monitor the SMS content provider and delete particular SMS according to the changes [78].

External Storage The file system of Android inherits that of the Linux kernel. The files are protected with read/write/execute permission for each user. Therefore, on non-rooted devices, we can only monitor the files or directories which are readable to third-party apps. For example, we cannot monitor system files under `/data/` directory, since the external apps don't have the read permission. External storage (known as the `/sdcard` directory) is a platform-specific file system module on Android, which is public to third-party apps. To access external storage, apps always need permission `READ_EXTERNAL_STORAGE` or

`WRITE_EXTERNAL_STORAGE`. In this work, we focus on monitoring external storage directory. Previous works use `FileObserver` to notify the file system changes [37].

Interrupt Statistics The logs of the interrupts raised to the kernel are also readable to third-party apps through a virtual file interface – `/proc/interrupts`. With this interface, users can obtain information about how many interrupts have been received by the CPU since booting. Previous work [28] uses this interface to infer user’s sensitive information, e.g., unlock pattern. We can use this interface to observe the use of different resources on the device, e.g., the camera, the Bluetooth, the NFC and so on. More details can be found in Section 4.4.

Network Network is another kind of resource for users to access during runtime. With `INTERNET` or `ACCESS_NETWORK_STATE` permission, apps can obtain the connection state, open URLs, or send/receive TCP/UDP packets. To monitor the network activities, previous work MopEye [74] leverages `VpnService` API to intercept all traffic initiated from apps on the devices. This API is designed for app developers to build VPN apps.

Although we can observe these resources to represent sensitive events on the device, identifying the apps that trigger the events is still a mystery. We integrate these observers to build a monitor for capturing the sensitive events on the devices and use machine learning techniques to identify the initiator of them.

4.2.2 Motivation

Table 4.1 lists existing works about dynamic analysis of sensitive behavior. We can observe that they need to use API calls, system calls, and other low-level events to reveal the underlying behavior of the target apps. For example, CopperDroid [64] observes and dissects the system calls made by an app to reconstruct the behavior, e.g., file operations. A majority of these works are based on in-lab running envi-

ronment, including VM (Virtual Machine)-based emulators, modified OS/Android internals, and rooted devices.

Table 4.1: Existing tools for analyzing sensitive behavior of Android apps

Tool	Platform	Features
DroidScope [77]	QEMU based Emulator	API call, system call, Dalvik instruction and so on.
CopperDroid [64]	QEMU based emulator	API constructed from system call
VetDroid [82]	Modified system	API call
M. Karami et al. [40]	Any platform with instrumented App	System call
DroidBox [43]	Modified system	API call

Relying on the in-lab running environment, previous work requires input generation tools [35, 46] to automatically run the target apps. However, the event series generated by these tools cannot match the logic of mobile apps which is usually complicated, e.g., most apps require registration following strict commands. Choudhary et al. present that the maximum coverage of popular input generator tools is only 40% even with sufficient time for running the apps [20]. Our intuition is that humans may be more successful in invoking the relevant functionalities of apps, and thus can achieve better code coverage with enough time and a large number of users. On another hand, app developers, especially those who design malware, would not prevent their apps' behavior from being triggered under real execution environments. Hence, deploying dynamic analysis to public users for crowdsourcing solves the code coverage problem.

Besides the coverage problem, running on emulators cannot analyze some environment sensitive apps. Petsas et al. [54] proposed a range of techniques to evade dynamic analysis in the emulated Android environment. With these techniques, apps can bypass the analysis of the tools, e.g., CopperDroid and DroidScope.

In this chapter, we introduce our dynamic analysis system named UpDroid. It gathers data from users' daily running traces and generates sensitive behavior reports for the apps.

4.3 System Overview

Figure 4.1 shows the framework of UpDroid, which consists of two major components: the monitoring module on the users’ devices and the analysis module on the server side.

We place two monitors on Android devices to monitor sensitive behavior (e.g., accessing the camera) and collect runtime status (e.g., CPU usage) of running apps. The data collected by both monitors is logged with time stamps. The event monitor detects changes to resources which can be accessed by third-party apps, e.g., the file system, to reveal the apps’ behavior. We choose to use these changes to represent the sensitive behavior as low-level events are not accessible on non-rooted devices.

However, we cannot identify the initiating app for the detected events without penetrating to apps or the systems. Hence, in the analysis module, we build an app identification model with machine learning techniques to distinguish the initiating app from all running apps. We use learning to rank to train the model with data from real users as presented in Section 4.5. We take sensitive events and the corresponding runtime status of the apps as inputs to identify the initiating apps for the monitored events and generate behavior reports for the apps. In the following two sections, we will present the technical details of the event monitoring and how we identify the initiating app for each event.

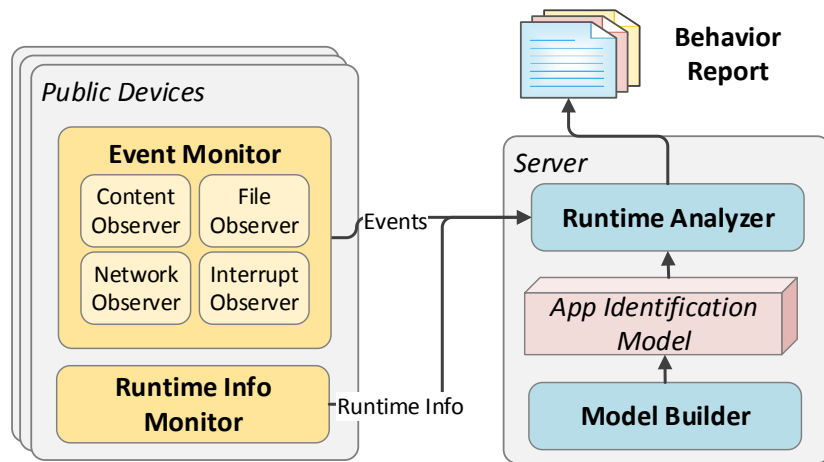


Figure 4.1: Framework of the sensitive behavior monitoring system - UpDroid.

4.4 Event Monitoring

This section describes how UpDroid monitors sensitive events without penetrating to either the apps or the Android internals on non-rooted phones. To reveal the behavior of the apps, UpDroid passively captures the events triggered by sensitive behavior. Figure 4.2 presents four types of sensitive behavior that can be monitored by UpDroid. The behavior is categorized by the resources, e.g., the file system, it manipulates. For example, accessing the camera raises a particular interrupt, so it belongs to the interrupt-based behavior. We use different methods to monitor different categories of behavior.

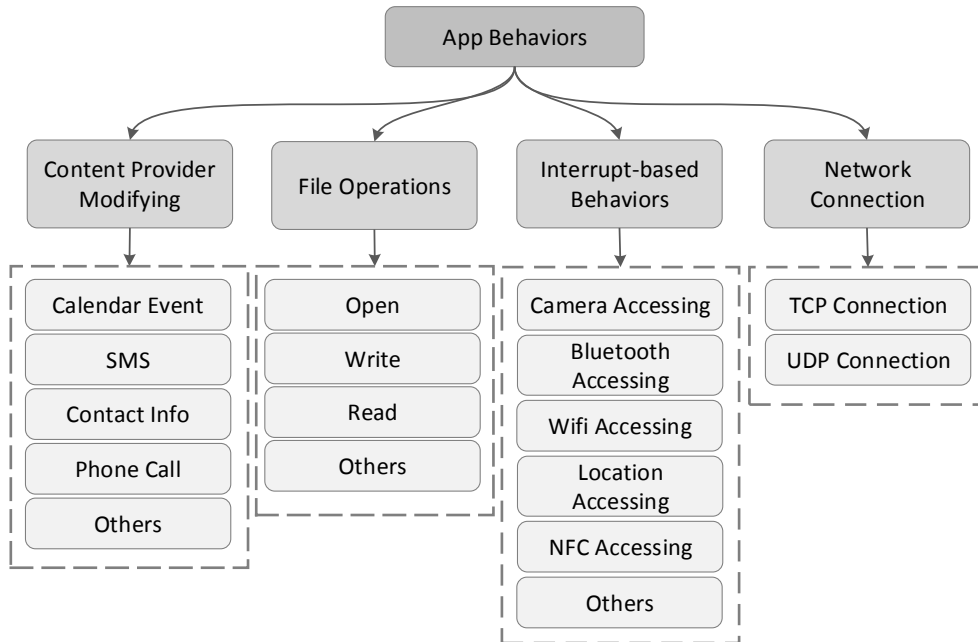


Figure 4.2: Different categories of behavior that UpDroid can monitor

4.4.1 Content Observer

UpDroid uses the `ContentObserver` API to capture the behavior that changes content providers. The method `registerContentObserver(Uri uri, boolean notifyForDescendants, ContentObserver observer)` is used to register a content observer with the corresponding URI, e.g., `content://sms` for observing SMS content. When an event is detected,

the `onChange()` method will be triggered. The `ContentObserver` only reports whether a content provider is changed, but not what has been changed. Hence, we log the monitored content provider and compare the updated provider with that at a previous timestamp after receiving a change notification, in order to get detailed information for inferring the apps' behavior. For example, row adding of SMS content provider with entry `type=0` represents sending out an SMS, and row adding with entry `type=1` represents receiving an SMS. The entries for each row also provide information, such as when the SMS is sent and the recipient of the SMS. To find all observable content providers, we use `PackageManager` to list all providers which can be accessed by external apps. For each provider, we query the corresponding database to find all table names which is also used as the path prefixes of the URIs of providers. From Android 6.0.1, we find 21 system provided content providers. Axplorer [7] can also identify the system content providers that are protected by permissions, but it does not filter out the ones that can only be accessed by the system apps. Theoretically, UpDroid can monitor all content providers with the required permissions. However, due to the overhead of logging and comparing the content providers, UpDroid only observes four of the most common and significant content providers, including SMS, call log, contacts and calendar events.

4.4.2 File Observer

To monitor events related to the file system, UpDroid uses `FileObserver` API to monitor the files and directories on the external storage. This API is provided by Android to capture changes to a single file or a directory. Event masks (e.g., `CREATE`, `DELETE` and `MODIFY`) are used to specify what kind of operation has been performed on the monitored file or directory. A complete list of the event masks can be found in Android API reference [34]. The `onEvent()` method will be triggered when an event to the file or directory is observed. Since this API

only supports single file or directory monitoring, we recursively traverse the monitored directory and register file observer for each file or directory under it. Similar to `ContentObserver`, `FileObserver` only reports the events but not the changing content. For example, `FileObserver` does not report how the file is modified when it captures a `MODIFY` event. Backing up the target directory is a possible solution to get detailed information about the events, but this may bring in too much space and runtime overhead. Hence, UpDroid only observes different types of events to the files in the external storage and ignores the detailed changes to them.

4.4.3 Interrupt Observer

A novel method we propose for observing events is to sample the interrupts and monitor the changes of interrupt numbers. Android inherits the interrupt mechanism of Linux. Interrupt represents the situation where CPU interrupts the running program to handle a request raised by an external hardware device. When the devices (e.g., camera, Bluetooth and temperature sensor) detect physical events, they raise interrupt requests. Then, the programmable interrupt controller (PIC) will process these requests and send them to the CPU. The CPU will finally respond to the interrupt requests.

Each specific interrupt will be registered to the system with a unique Interrupt Request Line (IRQ) number, through which devices can pass the interrupt to the processor. The virtual file `/proc/interrupts` provides the interrupt request lines claimed by the devices. Each line shows the unique IRQ number, the number of interrupts handled by each CPU, the PIC, and the device name.

To identify the events from the number of interrupts, we sample the `/proc/interrupts` file each 100ms and compare it with the previous sampling. Since most hardware devices have a corresponding IRQ line, we can infer the running status of hardware through monitoring the changes to the numbers of

the interrupts. The increases of the interrupts represent the sensitive behaviors. For example, accessing the camera increases the number of interrupt number 83 on Nexus 6P. Using Bluetooth to send a file to another device will increase interrupt 503 continuously for a period. In UpDroid, we choose to monitor the following five common devices: camera, GPS, Bluetooth, NFC, and video decoder.

Most device names shown in `/proc/interrupts` are coded with the hardware model names or abbreviations, thus are difficult to identify. For example, on HUAWEI Nexus 6P, `pn548` is the interrupt name of NFC and `atmel_mxt_ts` is for the touchscreen. Moreover, there are different IRQ lines with the same device name. For example, IRQ numbers 83 to 86 have the same device name `csid`, but only number 83 represents the camera device interrupt. Hence, the interrupt to hardware device mapping relies on the model names of the hardware devices which are difficult to obtain automatically.

Each mobile device has its own mapping between interrupts and hardware devices. Hence, we need to test the hardware devices and analyze the interrupt sampling to identify the interrupt mapping for each device model. We first analyzed several devices we already have, such as Nexus 6 and Nexus 6P. To cover other devices, we conduct a user study (named *interrupt study*) to obtain the mappings for them. Each UpDroid user needs to finish this study to get the interrupt mapping. The users need to perform certain operations to test the hardware devices on the phones. Some of the hardware devices can be tested automatically with proper programming. For example, we write a program to open the camera and take photos automatically. The others require the users' manual tests since the permission of these devices are very strict. For instance, NFC can only be manually turned on/off by the user for security consideration of Android. While the user is performing the tasks, the monitoring app samples the interrupts. By observing the changing pattern of the interrupts, we can identify the one that corresponds to the hardware devices. For each hardware, we need five traces for manually identifying the interrupt patterns. From the recruited participants, we have identified interrupts for 19 Android

devices.

4.4.4 Network Observer

UpDroid monitors the networking behavior through `VpnService` API, which leverages the TUN virtual network device to capture the TCP and UDP packets sent by the apps. In this work, we leverage MopEye’s technique to identify the package initiators through the `proc` file `/proc/net/tcp6|tcp|udp|udp6` [74].

4.5 Initiator Identifying

The monitoring techniques presented in the previous section are not able to identify the app that triggers the detected event since the APIs used by the monitors are designed for observing the resources. Although MopEye [74] provides an intelligent solution for app identification on network events, it is not applicable to other events, e.g., interrupt-based events. Hence, UpDroid leverages a machine learning technique, learning to rank, to build an app identification model which is generic for all events. This model takes the detected event and the runtime information of the running apps as input, and ranks all the running apps to find out which is the one that initiates the detected event.

The overview of the model learning is presented in Figure 4.3. To get the ground truth for the model learning, we recruit Android users as the inspectors to identify the apps that trigger the detected events. After pre-processing the data from the inspectors and the monitors, we use the learning to rank technique to train the identification model with the feature vectors.

4.5.1 App Status Monitoring

The runtime info monitor on the device collects information (e.g., CPU usage) about the running apps. We use the information as the feature of each app to infer whether

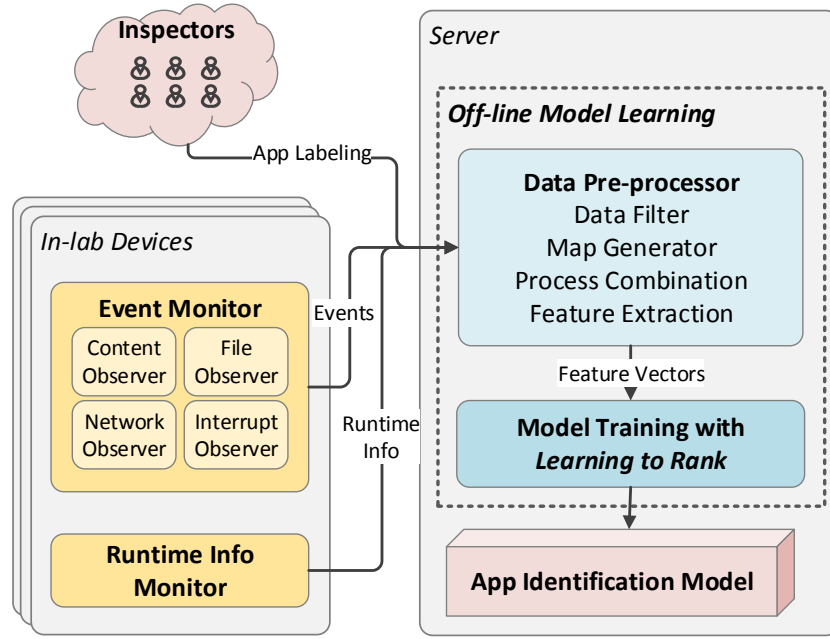


Figure 4.3: Overview of building the app identification model

it is the one that invokes the detected event. UpDroid uses the `ps` command to obtain the runtime information of the apps. It leverages the `/proc/stat` and `/proc/$PID/stat` interfaces to provide processes' runtime status, such as CPU usage, NICE value, virtual memory usage and so on. It allows third-party apps to access other processes' runtime info on most Android devices. We obtain the result from `ps` command from the monitoring app each 100ms. This time interval ensures both the quality of the data and the performance of the device.

One problem with using `ps` command is that the runtime status is for processes while identifying the process for a detected event is nearly impossible for users. For each occurred event, normal users can easily select the correct app that invokes the event, but can hardly tell which process without any knowledge about the app implementation. In this case, we consider *app* rather than *process* as the initiator of the detected events. We integrate the runtime status from processes from one app and get the ground truth of the initiators from the inspectors. More details of integrating runtime info from different processes can be found in Section 4.5.3.

4.5.2 Data Collecting

We collect the data for building the app identification model from the monitors presented in Section 4.4 and Section 4.5.1. The data contains the events and the runtime information of each app when an event occurs. The missing part is the ground truth that which one among all the running apps invokes the detected event. To get the ground truth, we recruit Android users to help to collect the data and label the app in real-time.

We gather three types of information from the inspectors' devices - the events, the runtime info, and the initiating apps selected by the inspectors.

We conduct a user study (named *initiator study*)¹ to collect and label the data. In this study, each participant will be asked to install our monitoring app published on Google Play and help to identify the apps at runtime. The monitor will capture the events and log the runtime app info. It would raise notifications to the participants (or inspectors) when it captures an event on the device. For each event, we provide the event type, the event content and the time when the event is captured to the participant. The participant responds to the notification and chooses the app that invokes the event based on the provided info. To check the integrity of the data, we propose the following policies to verify the participants' responses:

1. The selected app should be on the list of the running apps.
2. The selected app should have the permission for the detected event, e.g., the app chosen for a *camera* event needs to be granted with the `CAMERA` permission.
3. The selection should be finished within ten mins after the notification, to make sure the user selects with a fresh memory.

We have recruited ten users since November 2017 to participate in the initiator study for data collection and labeling. The participants are Android phone users

¹Both the interrupt study and the *initiator study* have been approved by IRB in May 2017

above 18 years old. Participants install the inspecting app on their own devices and identify initiators for at least 20 detected events. During the study, participants need to have at least ten apps they commonly use installed on their devices. We use the above policies to filter the responses from these participants. In total, we have collected 300 events with initiator identified. The initiators of these events correspond to 40 popular apps, e.g., Instagram and WhatsApp.

4.5.3 Data Pre-processing

From the monitors and the inspectors, we obtain the raw data for the analysis, including the detected events, runtime information of apps and the app identified by the inspectors. To get the labeled data, we pre-process the raw data in three steps.

First, we obtain runtime app info which can represent the running apps' status for each detected event. We choose the nearest samples of runtime info before and after each detected event based on the timing info. Hence, for each event, we know the apps' current state and how it changes after the event occurs.

Table 4.2: Features for running apps and the process combination rules.

Feature	Description	Type	Combination Rule
VSIZE	Size of virtual memory used	Integer	average
RSS	Resident set size	Integer	average
CPU	CPU usage	Integer	average
SCHED	Schedule of the process	Integer	average
PRIO	Priority	Integer	average
NICE	Nice value	Integer	average
PCY	Background/Foreground Info	Binary	or
PC	Status of processes	Binary	or
UID	Whether the app is system app	Bool	none

Then, we combine runtime info from different processes of an app.

As mentioned in Section 4.5.1, we collect the apps' runtime information by `ps` command which provides information about each running process, while *app* is the analysis target. Hence, we combine the runtime information of different processes from the same app.

Table 4.2 shows the features we collected for each app and how they are combined from different processes. We use the difference in runtime information before and after the event as the feature vectors.

Here is an example of extracting feature f_1 for an app. App a has two processes: p_1 and p_2 . Event e is observed at time t . The process sampling provides the nearest process info logs at time t_1 and t_2 , while $t_1 \leq t \leq t_2$. The f_1 value of p_1 is v_1 at t_1 and v_2 at t_2 . The f_1 value of p_2 is u_1 at t_1 and u_2 at t_2 . Hence, the processed feature f_1 of a for e is:

$$f_1 a = AVG(v_2 - v_1, u_2 - u_1)$$

Lastly, we identify the app that triggers it from the users' responses and label all running apps. For each event, we label "1" if an app is selected for it and label "0" if it is not selected.

4.5.4 Modelling and Precision

The machine learning technique we use for identifying the app is learning to rank [15]. Our scenario is a ranking problem, where we need to select an app that has the highest possibility of invoking the event among a list of running apps.

We use RankLib [23], a library that contains several popular ranking algorithms, for the modeling and testing. The model built by RankLib is generic for all apps and all events.

We randomly pick two-thirds of the data samples as the training data and the rest as the testing data. We tried all of the eight algorithms in RankLib with different configurations and compared their performance. Figure 4.4 presents the precision, the percentage of events the initiator of which can be successfully identified, of the models built by different ranking algorithms. With the LambdaMART algorithm, the precision of UpDroid can reach 80%, and the false alarm (the situation where the app ranked first does not cause the event) rate is around 20%. From our observation,

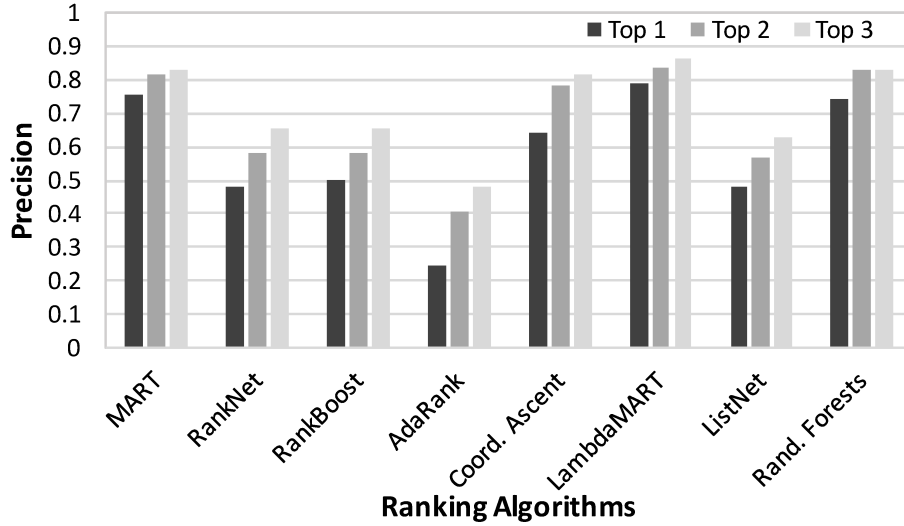


Figure 4.4: The performance of different ranking algorithms in RankLib library.

in one data sample, different apps may have the same ranking score, and this brings in a lot of false alarms. Hence, we list the apps with top1, top2 or top 3 ranking scores to see whether they contain the one selected by the inspectors. The precision comparison of the eight algorithms is presented in Figure 4.4.

4.6 Comparison with API hooking

Various tools [43, 64, 82] analyze sensitive APIs to reveal the underlying behavior of the target apps. For example, `sendTextMessage()` reveals the behavior of sending SMS. These tools log the APIs called by an app and record their parameters and return values by hooking all sensitive APIs, which typically requires root permission or modification to Android internals. In this section, we present the comparison between UpDroid and the API hooking method on capturing the sensitive behavior.

4.6.1 Current State of API hooking

To analyze sensitive behaviors through API hooking, we need a list of sensitive APIs and the permissions they require to define the behavior. Sensitive API refers to the API protected by certain permission. Since 2011, researchers have stud-

ied to extract the list of sensitive APIs from Android source code [6, 7, 12, 30]. From existing works, static analysis, including code analysis and annotation analysis, is believed to be the most efficient and effective method. However, from our investigation, none of the current methods can provide a complete list of the sensitive APIs. The popular tool Axplorer [7] provides an accurate list of the Android APIs in the Android framework source code, but it misses the analysis of Java APIs and the APIs whose permission checking is in native code. Some popular sensitive APIs, such as `android.hardware.camera2.CameraManager.openCamera()`, are not in the list. DPSPEC [12] analyzes the annotation of Android source code to identify sensitive APIs. However, it only focuses on the APIs protected by dangerous permissions and needs manual identification to obtain the list. Another work, android-a2p [30], is also based on annotation analysis. It is released on GitHub and is accurate but not complete. In order to cover more APIs for capturing a complete list of sensitive behaviors, we combine the lists from these three works in the comparison.

API hooking is also effective in capturing accessing sensitive content providers and passing sensitive Intent. DPSPEC and Pscout [6] list the content providers and intents which need dangerous permissions. We also include these sensitive components in our comparison.

In this work, we use an open source tool named EagleEye [49] which is built on the Xposed framework [76] to hook the sensitive APIs and the APIs for accessing content providers and sending intents on a rooted device.

4.6.2 Permission Coverage Comparison

To see how far UpDroid can go in covering different categories of events, we analyze the permission coverage of UpDroid and that of API hooking (including hooking sensitive APIs, content providers, and Intents). We use 26 *dangerous* permissions and 44 *normal* permissions crawled from Google’s official documentation for

Table 4.3: The comparison of dangerous permission coverage between API hooking and UpDroid. In this table, \times stands for none of the permissions in this categorize is covered and \checkmark stands for all are covered.

Permission	API Hooking			UpDroid
	Sensitive API	Content Provider	Intent	
ACCESS_COARSE_LOCATION	\checkmark	\times	\checkmark	\checkmark
ACCESS_FINE_LOCATION	\checkmark	\times	\checkmark	\checkmark
ADD_VOICEMAIL	\times	\checkmark	\times	\checkmark
ANSWER_PHONE_CALLS	\times	\times	\times	\checkmark
BODY_SENSORS	\times	\times	\times	\times
CALL_PHONE	\checkmark	\times	\checkmark	\checkmark
CAMERA	\checkmark	\times	\checkmark	\checkmark
GET_ACCOUNTS	\checkmark	\times	\times	\times
PROCESS_OUTGOING_CALLS	\times	\times	\checkmark	\checkmark
READ_CALENDAR	\times	\checkmark	\times	\times
READ_CALL_LOG	\times	\checkmark	\times	\times
READ_CONTACTS	\checkmark	\checkmark	\times	\times
READ_EXTERNAL_STORAGE	\checkmark	\checkmark	\times	\checkmark
READ_PHONE_NUMBERS	\times	\times	\times	\times
READ_PHONE_STATE	\checkmark	\times	\checkmark	\times
READ_SMS	\checkmark	\checkmark	\times	\times
RECEIVE_MMS	\checkmark	\times	\checkmark	\checkmark
RECEIVE_SMS	\times	\checkmark	\checkmark	\checkmark
RECEIVE_WAP_PUSH	\times	\times	\times	\times
RECORD_AUDIO	\times	\times	\times	\times
SEND_SMS	\checkmark	\checkmark	\checkmark	\checkmark
USE_SIP	\checkmark	\times	\times	\times
WRITE_CALENDAR	\times	\checkmark	\times	\checkmark
WRITE_CALL_LOG	\times	\checkmark	\times	\checkmark
WRITE_CONTACTS	\times	\checkmark	\checkmark	\checkmark
WRITE_EXTERNAL_STORAGE	\checkmark	\checkmark	\times	\checkmark
Total \checkmark	21/26			15/26

the comparison. If any sensitive API in the list uses certain permission, we consider that API hooking covers this permission. Also, if UpDroid can capture one kind of behavior which is protected by a permission, we consider that UpDroid covers this permission. This comparison may have inaccuracy since neither sensitive API nor UpDroid cover all the cases that a permission is used. However, this analysis still gives us a hint about what kind of behavior UpDroid and API hooking can capture.

The comparison of dangerous permission coverage is presented in Table 4.3.

The list of sensitive API/Content Provider/Intent we obtain from previous works covers 21/26 *dangerous* permissions, while UpDroid covers 15/26. And 14 permissions can be covered by both methods. As presented in Table 4.3, most of the ones

that cannot be captured by UpDroid are about reading data, e.g., `READ_CALENDAR` and `READ_PHONE_NUMBERS`. This is because UpDroid focuses on the behavior that changes the state of resources on the device while reading normally does not cause any change to them. The comparison of normal permission coverage can be found in Table 4.4.

4.6.3 Event Details Comparison

API hooking can provide details about each event or behavior based on the parameters and return value of an API, while UpDroid has a different method of providing detailed information for each event. We present the difference of monitoring each category of events as follows.

Content Provider:

While observing content providers, the events come from the changes to the providers. The detailed information of the events can be obtained from changes to the rows in the provider's table.

In the case of SMS activities, API hooking logs the `sendTextMessage()` API in apps, while UpDroid observes the `content://sms` content provider. Table 4.5 shows the information we can obtain from API hooking and content observing. Hooking APIs can get more low-level information, such as the Intent for sending this SMS. UpDroid can get more general information, such as when the SMS request is generated and when the SMS is successfully sent out.

Interrupt:

The monitoring based on interrupt sampling observes events from the changing of the number of the corresponding interrupt. It tells whether an event occurs and when it occurs through the changing pattern. Take Bluetooth interrupt as an example. Tracing API `android.bluetooth.BluetoothAdapter.enable()` can detect turning on of the Bluetooth on the devices. On the other hand, UpDroid observes it through recognizing a steep increase of the interrupt. Using Bluetooth

Table 4.4: The comparison of normal permission coverage between API hooking and UpDroid. In this table, ✕ stands for none of the permissions in this categorize is covered and ✓ stands for all are covered.

Permission	API Hooking			UpDroid
	Sensitive API	Content Provider	Intent	
ACCESS_LOCATION_EXTRA_COMMANDS	✓	✕	✕	✓
ACCESS_NETWORK_STATE	✓	✕	✕	✕
ACCESS_NOTIFICATION_POLICY	✕	✕	✕	✕
ACCESS_WIFI_STATE	✓	✕	✕	✕
BLUETOOTH	✓	✕	✓	✓
BLUETOOTH_ADMIN	✓	✕	✓	✓
BROADCAST_STICKY	✓	✕	✕	✕
CHANGE_NETWORK_STATE	✓	✕	✕	✕
CHANGE_WIFI_MULTICAST_STATE	✕	✕	✕	✕
CHANGE_WIFI_STATE	✕	✕	✕	✕
DISABLE_KEYGUARD	✓	✕	✕	✕
EXPAND_STATUS_BAR	✕	✕	✕	✕
GET_PACKAGE_SIZE	✓	✕	✕	✕
INSTALL_SHORTCUT	✕	✕	✓	✕
INTERNET	✓	✓	✕	✓
KILL_BACKGROUND_PROCESSES	✓	✕	✕	✕
MANAGE_OWN_CALLS	✕	✕	✕	✕
MODIFY_AUDIO_SETTINGS	✓	✕	✕	✕
NFC	✓	✕	✓	✓
READ_SYNC_SETTINGS	✓	✓	✕	✕
READ_SYNC_STATS	✓	✕	✕	✕
RECEIVE_BOOT_COMPLETED	✓	✕	✓	✕
REORDER_TASKS	✓	✕	✕	✕
REQUEST_COMPANION_RUN_IN_BACKGROUND	✕	✕	✕	✕
REQUEST_COMPANION_USE_DATA_IN_BACKGROUND	✕	✕	✕	✕
REQUEST_DELETE_PACKAGES	✕	✕	✕	✕
SET_ALARM	✕	✕	✓	✕
SET_WALLPAPER	✓	✕	✕	✕
SET_WALLPAPER_HINTS	✓	✕	✕	✕
SIGNAL_PERSISTENT_PROCESSES	✕	✕	✕	✕
TRANSMIT_IR	✓	✕	✕	✕
USE_FINGERPRINT	✓	✕	✕	✕
VIBRATE	✓	✕	✕	✕
WAKE_LOCK	✓	✕	✕	✕
WRITE_SYNC_SETTINGS	✓	✕	✕	✕
Total ✓	26/35			5/26

Table 4.5: The SMS event details provided by API Hooking and the content observer of UpDroid

Info	API	Content Provider
Destination Address	✓	✓
Source Address	✓	✓
Message Text	✓	✓
Sent Intent	✓	✗
Delivery Intent	✓	✗
Date Initiate	✓	✓
Date Sent	✗	✓
Person	✗	✓

continuously (e.g., sharing files) will be represented by a continuous slow increase.

External Storage:

As presented in Section 4.4, the information we can log from the file observers contains the file operation and the path of the file in external storage. To log the changes to the files, we need to back up the target files and make a comparison. To decrease the overhead, we choose only to record the operations and the paths. Existing API hooking method can hook file operation APIs, such as `java.io.writer.write(String s)`. It tells not only which operation is performed, but also the related content, e.g., the content that is written to a file.

Network:

In the case of network activities, UpDroid provides lower level information than API hooking, e.g., a TCP packet is sent by UID 10080 from 10.0.8.1:38175 to a server at 74.125.24.95:443. No higher level information, e.g., whether the packet is sent for loading a webpage, will be provided. The parameters and the type of the API imply the behavior of the app and the detailed information related to the behavior. It allows identifying different operations from the called API, e.g., `android.webkit.WebView.load(String URL)` represents loading a URL to a WebView.

4.6.4 Behavior Outcome Comparison

API hooking logs each attempt at using an API and needs further analysis to find out whether the called API is successfully invoked or not. Even with further analysis, it still misses the results of some app behavior. Contrarily, the four types of events reported by UpDroid represent the behavior that had successfully been performed. This is because it monitors the changes to public resources that will be manipulated by the apps' behavior. Here we compare the differences between UpDroid and API hooking in revealing the outcome of an attempt at performing a certain operation.

API hooking can use several ways to determine whether an API call is successfully called. The first and most apparent one is to check the return value. For example, `android.Bluetooth.BluetoothAdapter.enable()` returns boolean -“true to indicate adapter startup has begun, or false on immediate error”. The second way is to check the exceptions thrown by the API. For example, if `sendMessage` throws `IllegalArgumentException`, the message is not successfully sent because of empty destination address or text. Another more complicated method is to hook the callbacks as stated in the parameters. For example, `android.hardware.camera2.CameraManager.openCamera(String cameraId, CameraDevice.StateCallback callback, Handler handler)` has a parameter named `callback`. The callback will be invoked once the camera starts. For some other APIs, the callback is an intent which will be invoked after the API is successfully called. Comparing to the prior two methods, checking whether the API call succeeds or not through the third method needs more advanced API hooking techniques. These techniques should be able to obtain the callback from the API's parameter, hook it and determine whether the callback is invoked due to the API call. Hence, we only consider the first two methods in our analysis of the sensitive APIs.

From our analysis of the sensitive APIs, 154 out of the 400 do not have any implication about the result of the API call. And among the 154, there are 29

which use the permissions that can be covered by UpDroid. Among these 29, there are 14 that UpDroid conveniently reveals the outcome of the attempts. The rest is the APIs that do not change public resources on the devices. For example, `android.net.ConnectivityManager.requestNetwork()` requests network but does not send out packages, so the behavior cannot be detected by UpDroid. UpDroid can determine whether the behavior of the app changes the resources, but it cannot determine which API is used to trigger an event. UpDroid places more emphasis on the result of the app's behavior, while API hooking emphasizes more on the attempt of the app's behavior.

4.7 Capability Analysis

In this section, we evaluate the capabilities of UpDroid by analyzing its permission coverage and testing it on several popular apps. We also present the runtime performance of UpDroid evaluated with a popular benchmark app.

4.7.1 Permission Coverage

To evaluate whether UpDroid can detect the sensitive behavior that requires commonly used permissions, we analyze the permission usage of both malicious and benign apps. We analyze 2000+ malware samples (chosen from 72 malware families) provided by Android Malware Dataset Project [71] and 3000+ apps downloaded from the top chart of Google Play. For each permission, we count the number of apps that declare the permission in the manifest to find out the popular permissions used by malicious and benign apps. The dangerous permission usage is shown in Figure 4.5 and Figure 4.6. As presented, the permissions from `WRITE_CONTACTS` to `ANSWER_PHONE_CALLS` can be covered by UpDroid. The results show that UpDroid covers the widely used permissions. And it cannot cover the ones for reading private data or the phone states which will not cause any state changing of the observable resources on the device.

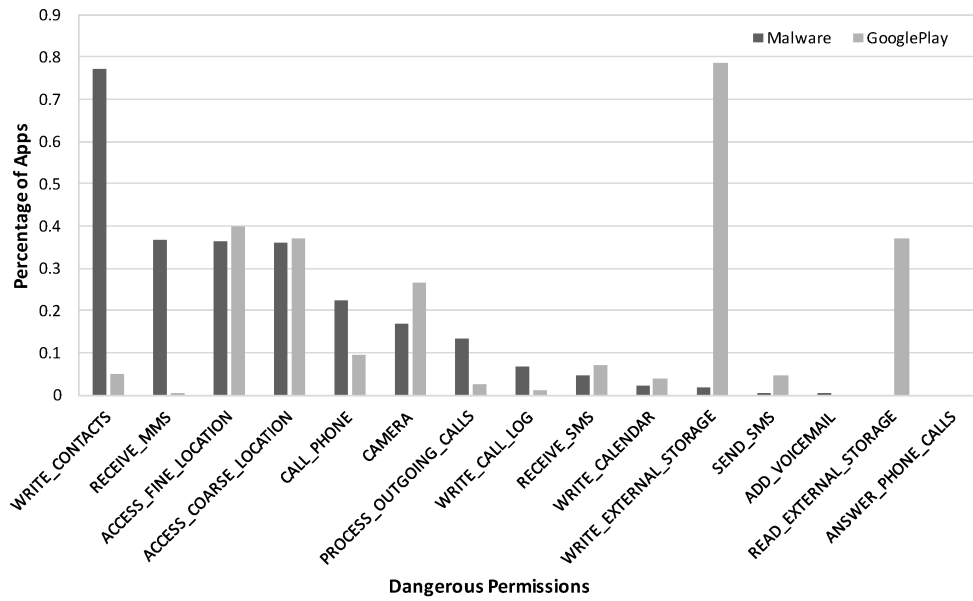


Figure 4.5: Dangerous permission usage (covered by UpDroid) of malware samples from AMD and benign apps from GooglePlay

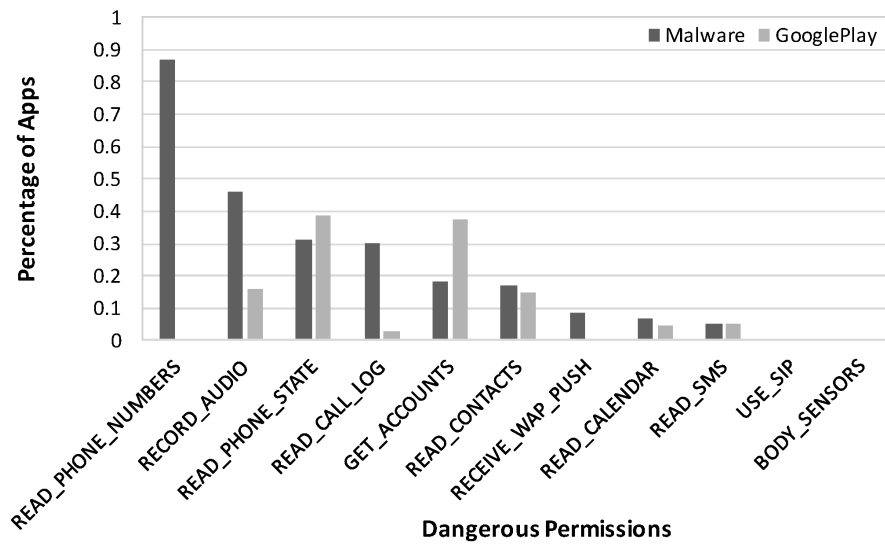


Figure 4.6: Dangerous permission (not covered by UpDroid) usage of malware samples from AMD and benign apps from GooglePlay

4.7.2 Runtime Experiments

To evaluate how UpDroid performs at runtime for capturing sensitive behavior, we test it on several popular apps, including a communication app WhatsApp, a social networking app Facebook, and an online shopping app Lazada. These apps have more sensitive behavior than most of the malware. We manually run the apps for five minutes while using UpDroid and API hooking to detect the sensitive behavior. We find that UpDroid successfully captures sensitive behavior, such as sending SMS, accessing the camera, opening Bluetooth and so on.

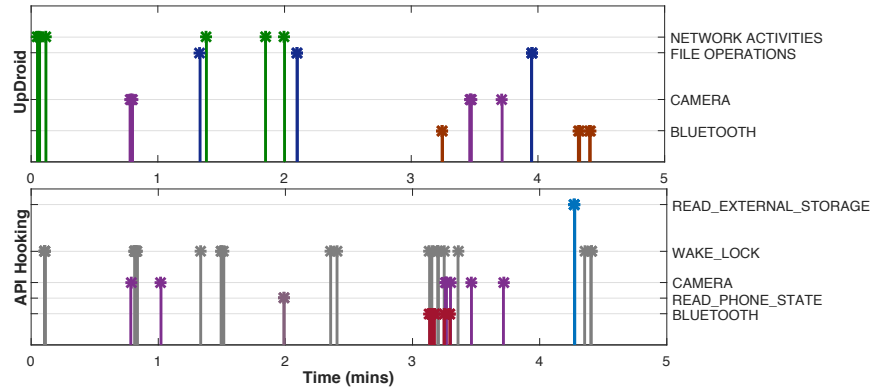


Figure 4.7: The runtime analysis results of WhatsApp from UpDroid and API hooking

Specifically, we present the experiment on WhatsApp which uses various permissions and compare the results (as shown in Figure 4.7) of UpDroid and API hooking. The upper figure presents the behavior captured by UpDroid, and the lower one presents the permission usage detected by API Hooking. The upper figure shows that UpDroid detects Bluetooth events, camera events, file operations and network activities of WhatsApp. Compared to API hooking, UpDroid detects the events which manipulate public resources on Android. For network events, UpDroid detects the packages sent out or received, while API hooking reports access to the network state. Although API hooking can also detect internet usage, no internet activity is found due to the incompleteness of the sensitive API lists. This also happens on the Bluetooth activities and file operations. As shown in Figure 4.7, using

Bluetooth is monitored by UpDroid at the fifth minutes, but it is not observed by API hooking. UpDroid captures multiple file system operations which are not detected by API hooking. It also shows that UpDroid cannot detect the read permissions like `READ_PHONE_STATE` and `WAKE_LOCK`.

4.7.3 Performance

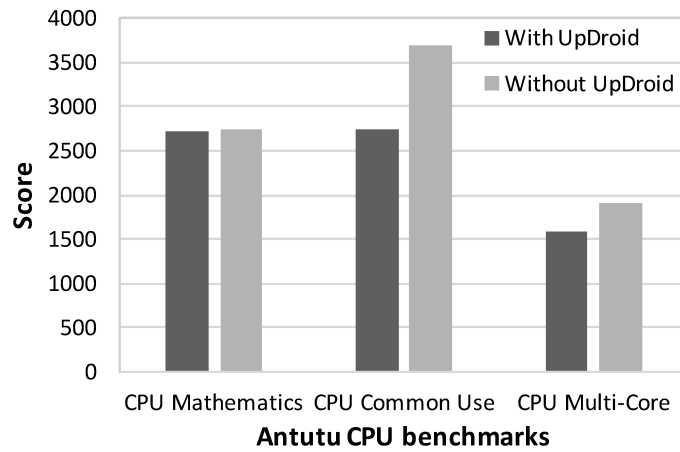


Figure 4.8: Performance of UpDroid evaluated with Antutu Benchmark

To evaluate the runtime overhead of UpDroid, we run the monitoring module of UpDroid on Nexus 6P with Qualcomm Snapdragon 810 processor and 3GB RAM. We install ten popular apps on the device and keep three of them running in the background. We use one of the most popular benchmarks, Antutu Benchmark, to grade the device with and without UpDroid running on it. The result is presented in Figure 4.8. The y-axis is the score graded by Antutu. The higher the score is, the faster the CPU runs. In total, UpDroid decreases the benchmark score by 15%. The overhead mainly comes from the high sampling rate of the interrupt numbers and the frequent use of `ps` command. There is a trade-off between the accuracy and the performance. The evaluation is conducted with a device released in September 2015. We believe that the overhead can be decreased on more powerful phones.

4.7.4 Discussion

In this section, we discuss how UpDroid can avoid anti-analysis techniques and the possibility of using UpDroid as an attack technique. We also present the limitations of UpDroid.

UpDroid is a dynamic analysis system which is transparent to malware. The monitoring module of UpDroid is implemented with the APIs widely used by app developers (e.g., `ContentObserver` and `VPNService`). The anti-analysis techniques are difficult to evade UpDroid. Among all the APIs, `VPNService` is technically detectable but cannot be used by malware as an indicator, because VPN is widely used by mobile users who need a secure and private network. It is also quite popular among Chinese users for accessing blocked websites. Instead of detecting APIs commonly used by normal apps, malware tends to use heuristics which imply the running environment is under analysis (e.g., invalid IMEI number and abnormal GPS info). On the other hand, it would be an advantage when all malware stops its malicious behavior after detecting UpDroid on the users' devices.

UpDroid is designed for analyzing the underlying app behavior, but the techniques used can also be applied to maliciously monitor the users. The monitoring technique that uses `/proc/interrupts` can be applied to side-channel attacks, which monitor sensitive behavior on the device without any permission needed. The app identification model also starts a study to break the process isolation on Android.

The limitation of UpDroid is that it requires access to `/proc` file system which is protected by critical SELinux policies since Android 7. From Android 7, the `ps` command cannot access the process info of other processes. Hence, we need to identify other runtime info, e.g., time for launching the other apps, as the feature of each app in the future. Android 8 prevents third-party apps to access `/proc/interrupts`. Hence, UpDroid may not be significantly effective on devices with Android 8 but still works on a larger proportion of devices with prior Android ver-

sions. AppBrain shows that the market share of Android SDK versions prior to 8.0 is 95.4% in April 2018 [5].

4.8 Summary

In this chapter, we present our efforts on the dynamic analysis of app behavior under unmodified and non-rooted devices. We propose UpDroid - a system for dynamically monitoring Android apps' sensitive behavior. It uses different APIs to monitor Android system at runtime and leverages learning to rank technique to identify the initiator of the detected behavior. We use the permission coverage, the runtime experiments and the comparison with the traditional API hooking method to demonstrate the capabilities of UpDroid. The results show that UpDroid can detect sensitive behavior that manipulates the resources of the devices and identify the apps that trigger the behavior. This work is done and published in Wisec 2018.

Chapter 5

App Analysis with GPU Interrupt Timing Information

Among the interfaces which can provide app runtime information on non-rooted devices, `proc` file system provides an accessible channel with large amount of resources for app analysis. This chapter aims to further investigate the possibility of app behavior analysis on non-rooted devices with information provided by `proc` file systems. We present a side-channel-based method for analyzing app activities without rooting or modifying the systems from the attackers' point of view.

5.1 Introduction

Linux and Linux-based systems are widely used on servers, desktops, and mobile platforms. They provide a `proc` file system (*procfs*) as interfaces for obtaining information about processes and other system information. However, the information disclosed by *procfs* is often used by side-channel attacks [19], such as inferring keystrokes [80], TCP sequence numbers [55], and user identities [84]. Among the resources from *procfs*, `/proc/interrupts` provides the statistics of both hardware and software interrupts raised to the CPU which can be used to inferring hardware events on the device. In this chapter, we investigate the possibility of build-

ing the connection between software-level information and the interrupts statistics. Specifically, we investigate whether GPU interrupts can be used to infer app activities on computer devices.

Since the introduction of Xerox Alto in 1973, graphical user interface (GUI) becomes more and more popular, and is now becoming indispensable in modern computer devices. Even for Linux on which command line interface can fulfill all the tasks, GUI is necessary to fasten the user's work. Moreover, mobile systems based on Linux are heavily dependent on user interfaces. The display of GUI presents viewable activities on the devices which reflects users' interaction with the device. GPU plays an important role in creating images as output through display devices, during which interrupts will be generated by GPU to interact with the CPU.

GPU interrupts are often used to signal completion of graphics commands, vertical blanking events, or reporting errors [48]. During displaying frames, different sequences of GPU interrupts will be generated and the pattern of GPU interrupt timing series can be used to identify the content of the display. Figure 5.1 presents the GPU interrupt timing series of launching WeChat and Instagram on Nexus 6. In this figure, the y-axis presents the number of interrupts raised by GPU in every 50ms. Our observation shows that different activities (e.g., different apps to be launched in this example) correspond to different patterns of the GPU interrupt timing series. Based on this observation, we conduct side-channel attacks for inferring app activities.

We choose Android and Ubuntu, two of the most popular Linux-based systems, as the target platform to conduct side-channel attacks. The attacks include inferring launched app and inferring activities within apps, such as identifying the webpages opened. The core idea of the attacks is to classify the interrupt timing series with machine learning techniques. We implemented the attacks on both Android and Ubuntu for identifying different activities with different models. Results show that the proposed side-channel attacks are highly effective on both Android and Ubuntu. Evaluation of the identifying models shows that the classification precision of the

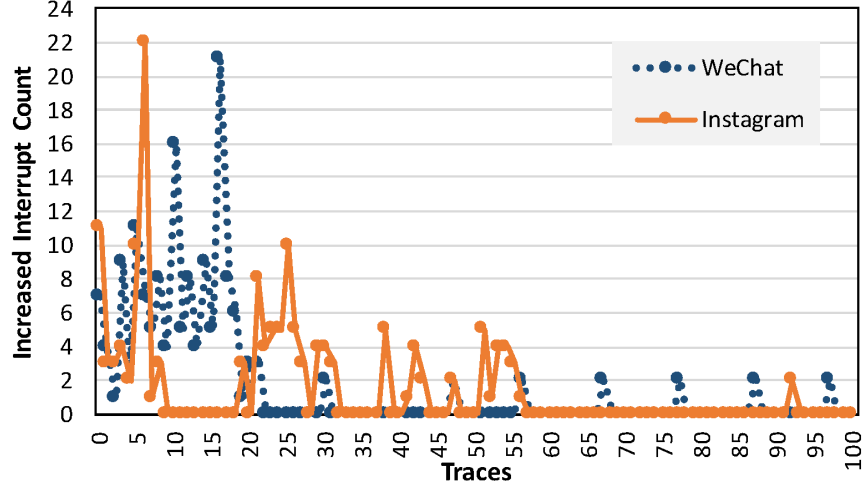


Figure 5.1: GPU interrupt increasing patterns while launching WeChat and Instagram on Nexus 6

launched app on Android and Ubuntu can reach 90% with less than 1 second sampling of the GPU interrupts. The classification for activities within an app can reach 80% with less than 2 seconds sampling. The experimental results show that GPU interrupt timing information can be used to infer app behavior with high precision and the side-channel attack we propose is highly effective on both Android and Ubuntu.


5.2 Background

In this section, we introduce background information for this work including the interrupt mechanism and GPU interrupt on Linux/Linux-based platform.

5.2.1 Interrupt Mechanism

On Linux/Linux-based platform, an interrupt is a signal emitted by hardware or software to CPU for suspending the current activities and processing specific event through the corresponding interrupt handler. The interrupt mechanism allows CPU to work with multiple hardware devices and enables the kernel to handle external events in time. During the initialization of a hardware device driver, the corresponding interrupt handler will be registered to the kernel. Upon receiving an interrupt,

the interrupt handler will perform specific actions to process the events. For example, when the keyboard device raises an interrupt to CPU, the CPU will stop the current activity and store the current context to read the inputs from the keyboard. After that, the CPU will load the stored context and resume the previous activity.



```

shell@angler:/ $ cat /proc/interrupts

```

	CPU0	CPU1	CPU2	CPU3	CPU4	CPU5	CPU6	CPU7		
20:	5185586	1855020	1110154	686421	299859	237093	217353	198063	GIC	arch_timer
35:	0	0	0	0	0	0	0	0	GIC	apps_wdog_bark
...
82:	897	822	71	2649	0	0	0	0	GIC	cci
83:	16	0	0	2	0	0	0	0	GIC	csid
84:	0	0	0	0	0	0	0	0	GIC	csid
85:	0	2	0	0	0	0	0	0	GIC	csid
86:	0	0	0	0	0	0	0	0	GIC	csid
87:	1185	0	0	0	0	0	0	0	GIC	
...
503:	0	0	4	0	0	0	0	0	msm_tlmm_irq	pn548
504:	3812	2989	2733	1878	0	0	0	0	msm_tlmm_irq	synaptics_dsx
506:	2098	4917	4530	1803	1127	101	2	0	msm_tlmm_irq	msm_pcie_wake
603:	41319	72786	21284	19259	114	26	0	0	msm_tlmm_irq	spi5.0
604:	10	2	0	1	0	0	0	0	msm_tlmm_irq	tusb320_int
605:	0	0	0	0	0	0	0	0	msm_tlmm_irq	bluetooth hostwake
606:	0	0	0	0	0	0	0	0	qnp-int	f98a4900.sdhci cd
...
IPI0:	5082009	6300589	4084595	3100888	1584977	629822	633236	503521	Rescheduling interrupts	
IPI1:	517060	1129677	1213071	1255469	34121	40161	42040	40424	Function call interrupts	
IPI2:	4133	119424	163757	99622	287310	175396	153678	141625	Single function call	
interrupts										
IPI3:	0	0	0	0	0	0	0	0	CPU stop interrupts	
IPI4:	224939	275760	205134	133385	32682	30935	30508	23840	Timer broadcast interrupts	
IPI5:	605720	261368	201074	165259	35295	9263	5534	5154	IRQ work interrupts	
IPI6:	0	18736	18736	18736	18748	18742	18755	18753	CPU wakeup interrupts	
IPI7:	0	0	0	0	0	0	0	0	CPU backtrace	
Err:	0									

Figure 5.2: A sample of the `/proc/interrupts` file on HUAWEI Nexus 6P

On the Linux operating system or systems based on the Linux kernel, the numbers of interrupts raised by different external hardware devices or software can be obtained through a virtual file interface - `/proc/interrupts`. Figure 5.2 presents a sample of the `/proc/interrupts` file on Nexus 6, a device based on Android system. As presented in this figure, the first column is the IRQ (Interrupt Request Line) number for each hardware or software interrupt. The following eight columns are the numbers of interrupts raised to different CPUs. The remaining columns show the interrupt controller names, e.g., GIC (General Interrupt Controller) and the device names, e.g., `arch_timer`. On Linux and other Linux-based systems, the `/proc/interrupts` has a similar structure.

5.2.2 GPU Interrupts

GPU is a hardware device which handles graphics computing and displaying. Normally, GPU will raise interrupts for signaling completion of graphics commands,

Table 5.1: The interrupt sources and description of NVIDIA GF119 [41]

Interrupt Source	Description
NVKM_ENGINE_DISP	the display engine
NVKM_ENGINE_MSPDEC	the picture decoder
NVKM_ENGINE_MSVDL	the variable length decoder
NVKM_ENGINE_GR	the memory copying, 2d and 3d rendering engine
NVKM_ENGINE_FIFO	the command submission to execution engine
NVKM_ENGINE_CE1(0)	the copy engine
NVKM_ENGINE_MSPPP	the video post-processor
NVKM_SUBDEV_IBUS	the intelligent BUS utility system
NVKM_SUBDEV_BUS	the bus controller
NVKM_SUBDEV_FB	the memory controller and arbiter
NVKM_SUBDEV_LTC	the LT controller
NVKM_SUBDEV_PMU	the power management unit
NVKM_SUBDEV_GPIO	the general purpose Input/Output controller
NVKM_SUBDEV_I2C	the i2c bus controller
NVKM_SUBDEV_TIMER	the GPU timer
NVKM_SUBDEV_THERM	the thermal sensor and clock throttling circuitry

vertical blanking, hotplugging and so on [?]. For example, table 5.1 presents the interrupt sources of NVIDIA GF119. By instrumenting the interrupt handler in **nouveau** driver, we found that the interrupts are mainly raised by the display engine, command submission to execution engine and the timer. The interrupts raised by the display engine are the vertical blanking signals when the displaying of a picture is completed. The number of interrupts raised by the display engine is the vertical refresh rate of the monitor. The command submission to execution engine (FIFO) gathers processing commands from the command buffers prepared by the host and delivers them to the memory copying and rendering engine in an orderly manner [41]. The interrupts raised by FIFO engine are the signals for command completion.

The `/proc/interrupts` file provides information about kinds of software and hardware interrupts. The software interrupts include interrupts raised by indirect calls, cache errors and other low-level behaviors of the system. The hardware interrupts include interrupts raised by different devices, such as bus, camera, bluetooth and GPU. Among all the interrupts, GPU is the component that would be used

by nearly all behavior on devices with GUI. It will be used in the displaying of the screen changes. The number of GPU interrupts represents the communication between CPU and GPU while the monitor or touch screen displays picture frames. Different app behavior leads to a different sequence of frames displayed, and the GPU interrupt number would be side-channel information for inferring app behavior. Hence, we propose to use the timing series of GPU interrupts as the feature of different activity. The assumption of this chapter is that different user activity leads to different screen displaying. In this chapter, we propose to infer app behaviors on the device by sampling the number of GPU interrupt in `/proc/interrupts` file and evaluate the precision this method can reach to support our assumption.

5.3 Methodology Overview

To disclose app activities without root permission or modified system, we log the increased number of interrupts raised by GPU when the behavior occurs and use the timing series of the GPU interrupt as a feature of the activity.

We implement a monitoring app to sample the number of the GPU interrupt raised. The application contains two main services. The first service runs in the background to read `/proc/interrupts` file and record the number of interrupts raised by GPU every 50ms. The second service records the most important user interactions related to app activities with the devices (e.g., touches on the screen). The timing of the interactions is used as the timestamps of the beginning of the activities. On Android, we record the touches on the screen by creating a one-pixel alert window on top of any other app. Third party apps are allowed to do so with permission `SYSTEM_ALERT_WINDOW` [67]. Although this method cannot provide either position or duration of the clicks, the timing information is enough for us to trigger interrupt sampling when a click is observed. On Ubuntu, we use the python library named *pymouse* to log the mouse clicks. As presented in Figure 5.1, different app activities correspond to different timing series patterns.

The methodology of inferring the activities is using machine learning techniques to build models for classifying interrupt timing series. We collect labeled data and train classification models for different scenarios, such as inferring the launched app. The tool we use for model training is Weka [36], which contains the most popular classifiers. The next section presents the experiments we conducted on Android and Ubuntu and the details about model building and evaluating.

5.4 Experiments on Android

In order to evaluate the feasibility of using GPU interrupt timing information to identify app behavior, we conduct three experiments on a Nexus 6 device.

5.4.1 Experiments Setup

The first experiment is for inferring the launched apps on Android. In this experiment, we download the top 20 apps on Google Play in September 2018 (see Figure 5.4 for the names of the apps). We use scripts to automatically launch each app 100 times. While clicking the app icon, the interrupt sampling begins and lasts for 5 seconds. The sampling rate is 20Hz which means we log the GPU interrupt amount with 50ms. Each time an app is launched, we will log the interrupt amount 100 times. In total, we will get a vector with 100 reading.

The second experiment is for inferring activities within an app. We choose WhatsApp as the test app and select 20 most popular activities in it. Each activity will start by a button clicking and will lead to the specific screen change. We automatically conduct those activities and log the interrupt at 20Hz for 5 seconds when the clicking happens. The maximum vector length is also 100 in this experiment.

The third experiment is for inferring the opened webpage with Chrome on Android. We choose eight popular webpages (see Figure 5.6 for the names of the webpages). Each webpage is loaded for 100 times with scripts and the interrupt is also logged at 20Hz for 5 seconds. For each webpage loading activity, we get a

vector at a length of 100.

For each experiment, we use Weka [36], a tool for classification and clustering, to classify different activities. We use 10 folder cross-validation to test the precision the classification can reach. The results are presented in the following section.

5.4.2 Model Precision

We select eight popular classifiers on Weka and classify the interrupt timing series in 5 seconds for the two kinds of activities (inferring the launched app and activities of WhatsApp). The length of the feature vectors is 100. Table 5.2 presents the classification precision for each classifier with 10 folder cross-validation. The best precision we can get for identifying the app launched is 92.70%, and 82.12% for identifying WhatsApp scenarios. Among these classifiers, BayesNet provides the best precision for identifying the launched app. RandomCommittee provides the best precision for identifying WhatsApp scenarios.

Table 5.2: Precision of different classifiers for inferring app activities on Android

Classifier	Precision (%)	
	App Launching	WhatsApp Scenario
Bagging	84.33	76.44
BayesNet	92.70	81.00
IB1	81.03	71.99
J48	75.36	73.96
LibSVM	59.53	70.01
NativeBayes	87.78	64.54
RandomCommittee	92.04	82.12
RandomTree	71.35	70.21

From the precision results, we can find that the precision of identifying the launched app is higher than inferring WhatsApp activities. To find out the reasons, we draw the heatmaps of the classification results for both experiments with the classifiers which provide the best precision. Figure 5.4 presents the heatmap of classifying launched app on Android. Generally, the precision is above 90% except for SHAREit. By observing the app launching process of SHAREit, we find that

SHAREit starts with a dynamic advertisement which leads to an unstable pattern of GPU interrupt timing series. Figure 5.5 presents the heatmap of classifying WhatsApp activities. We find that *WhatsAppSenario_1* and *WhatsAppSenario_2* are difficult to be differentiated. This also happens for *WhatsAppSenario_15*, *WhatsAppSenario_16*, *WhatsAppSenario_18* and *WhatsAppSenario_20*. By observing these scenarios, we found that the scenarios which are difficult to be differentiated have a similar layout. For example, *WhatsAppSenario_15* to *WhatsAppSenario_20* are the activities led by clicking the menu in WhatsApp. Most of them have a similar layout and style. Hence, the GPU interrupt timing patterns are similar for them, as presented in Figure 5.3

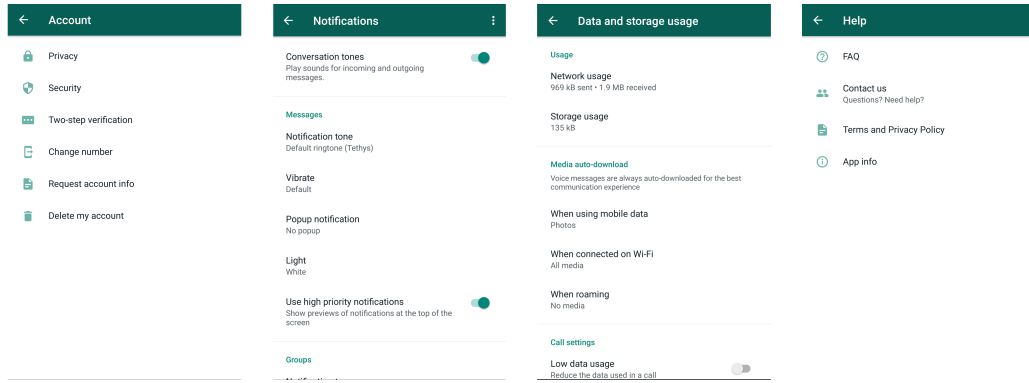


Figure 5.3: WhatsApp using scenarios that are difficult to be distinguished

We also presented the heatmap for classification results of inferring the loaded webpage on Android in Figure 5.6. The classification can reach 90% precision. The imprecision implies the similarity between webpages (e.g., *baidu.com* and *wikipedia.org* have indistinguishable instances).

We also use these two classifiers to evaluate how the time period affects the precision for inferring the activities. The results are presented in Figure 5.7. As presented in this figure, we can find that the smallest time period of sampling for reaching 80% precision of inferring the launched app is 2 seconds, while for inferring layout changes within WhatsApp is 1 second. This difference comes from the fact that launching an app is slower than changing a layout within WhatsApp on the experimental device.

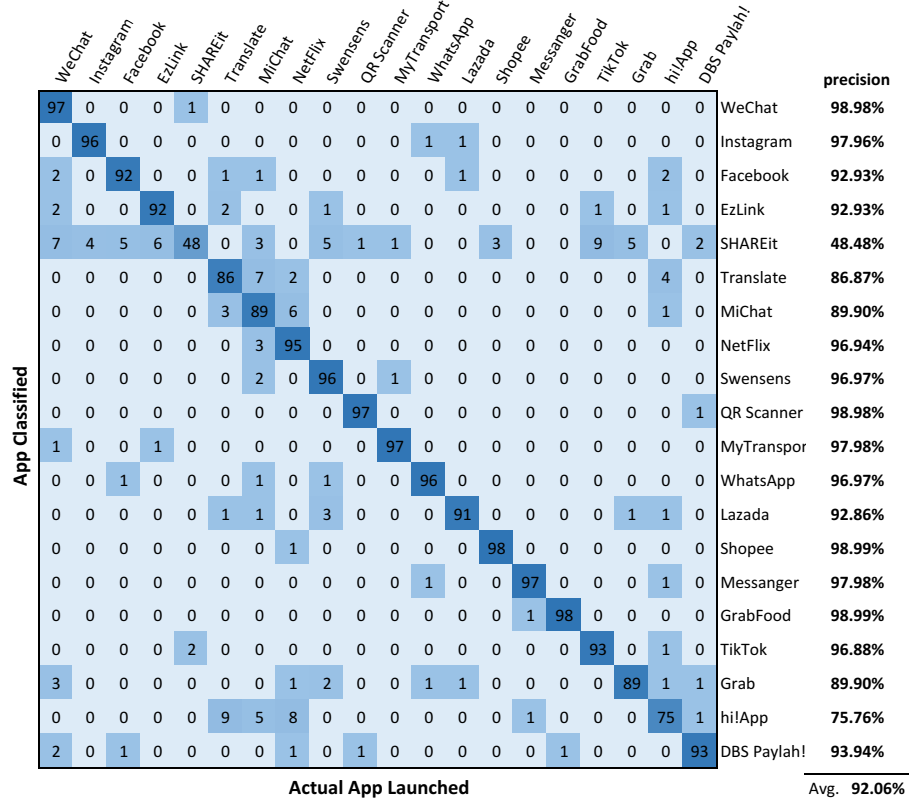


Figure 5.4: Heatmap for classification results of inferring the launched app on Android

5.4.3 Noise Analysis

In order to find out the strength of the trained models, we conduct an experiment in which running apps in the background are added as the noises to test the models. We start three popular apps (Chrome, Facebook and WhatsApp) and keep them running in the background. At the same time, Android also has its system apps and services running (e.g., nfc and bluetooth services). In total, there are 31 running processes and 12 running applications (including the google services) which are launched automatically. Meanwhile, we automatically launch WeChat and Instagram for 100 times to obtaining the timing series of GPU interrupt. We choose these two apps since WeChat has similar layout each time it is launched, while Instagram has different layout based on whether there is new post. The precision of identifying WeChat is 83.00% and 72.00% for identifying Instagram. The precision results shows that background apps slightly reduce the precision of the identifica-

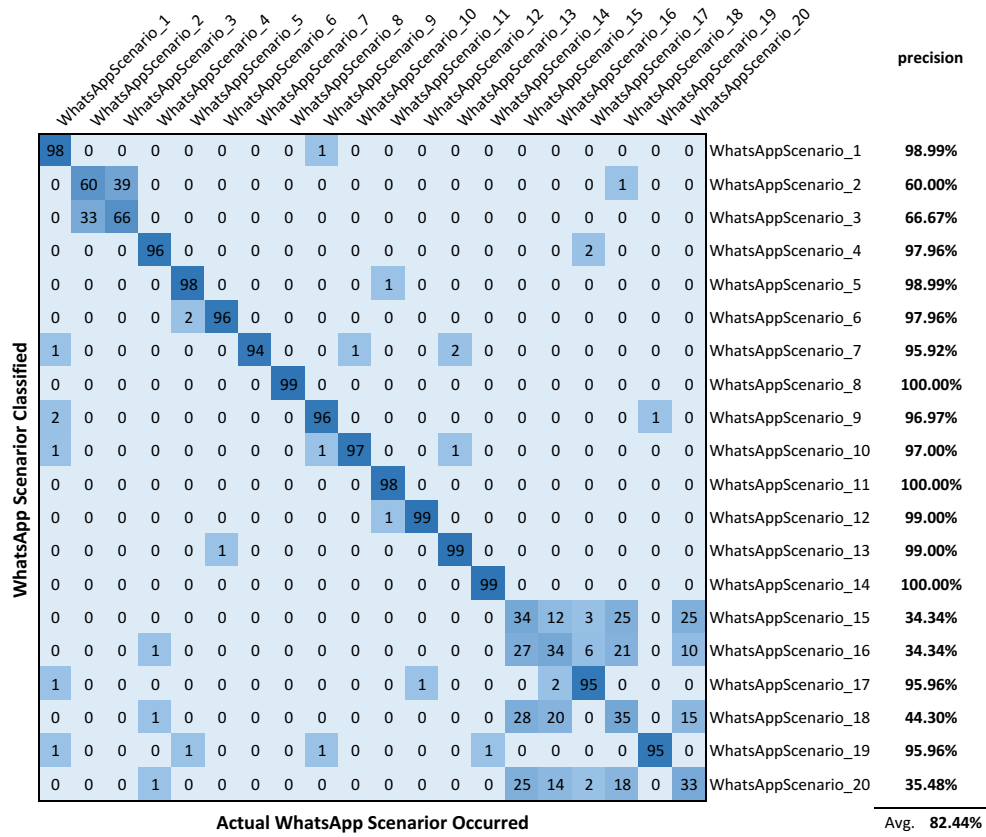


Figure 5.5: Heatmap for classification results of inferring the WhatsApp activities on Android

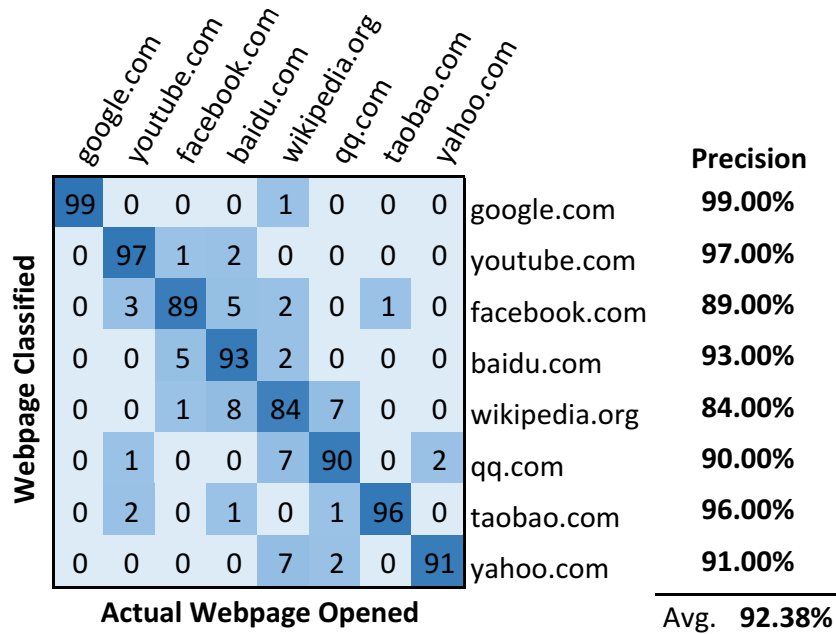


Figure 5.6: Heatmap for classification results of inferring the loaded webpage on Android

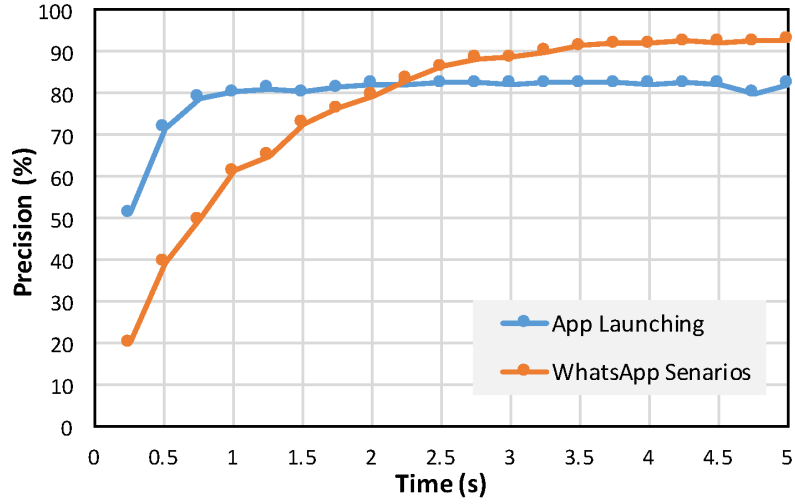


Figure 5.7: The precision of inferring app activities using different lengths of interrupt timing series on Android

tion models. To increasing the strength of the models, a possible solution would be adding background noises to the training set to simulating the real-world app using scenario.

5.5 Experiments on Ubuntu

We also conducted similar experiments on Ubuntu which are inferring the launched app and the opened webpage. The platform for this experiment is an Acer desktop device with Intel(R) Core(TM) i5-4590 CPU @ 3.30GHz and NVIDIA GPU GF119.

5.5.1 Experiments Setup

The first experiment is for inferring the launched apps on Ubuntu. In this experiment, we download 20 popular apps on Ubuntu, such as LibreOffice, vlc, Firefox and so on (see Figure 5.8 for the names of the apps). We use scripts to automatically launch each app 100 times. On Linux, it is easy to use the command line to launch apps, so we don't record the user interactions with the device. We log the timestamp of launching each app with the command line and log the interrupt amount every

50ms. Each time an app is launched, we will log the interrupt amount 100 times. In total, we will get a vector of the timing series for each app launching activity with a length of 100.

The second experiment is for inferring opening different webpages through chrome. We choose several popular webpages, such as Google, Facebook and so on. Interrupt amount is also logged every 50ms and 100 times for each webpage loading activity. In this experiment, each webpage will be automatically loaded for 100 times with Chrome.

For each experiment, we also use Weka [36] to classify different activities and use 10 folder cross-validation to test the precision the classification can reach. The results are presented in the following section.

5.5.2 Model Precision

Table 5.3: Precision of inferring app launching on Ubuntu using different classifiers

Classifier	Precision (%)	
	App Launching	Webpage Loading
Bagging	95.400	83.125
BayesNet	88.800	73.125
IB1	97.650	83.125
J48	96.150	80.000
LibSVM	31.750	22.125
NativeBayes	93.050	72.250
RandomCommittee	97.000	84.625
RandomTree	94.900	79.500

We use the same eight popular classifiers with Weka to classify the labeled timing series and use 10 folder cross-validation to evaluate the models. The results for inferring launched app and loaded webpage are presented in Table 5.3. The classification in this table uses feature vectors with a length of 100. The best precision for identifying the app launched is 97.00% and 84.625% for identifying the webpage opened. Similar to mobile, inferring app launching activities has higher precision than inferring loading different webpages in the same browser.

We also present the heatmap of classifying app launching and webpage loading activities in Figure 5.8 and Figure 5.9. As presented in Figure 5.8, all Ubuntu applications can be well classified with feature vectors in length of 100. Figure 5.9 shows that webpages loaded by the same browser can be classified with reasonable precision. Among the webpages, *google.com*, *baidu.com*, and *wikipedia.org* are not that well differentiated, since the layout similarity of them is higher than other webpages. Comparing to webpage identification on Android, the precision on Ubuntu is relatively lower. The reason is that the amount of information is similar on mobile and desktop, but desktop has a much larger screen size. The similarity of the displayed images is higher on desktop. Hence, the differentiation is more difficult on desktop.

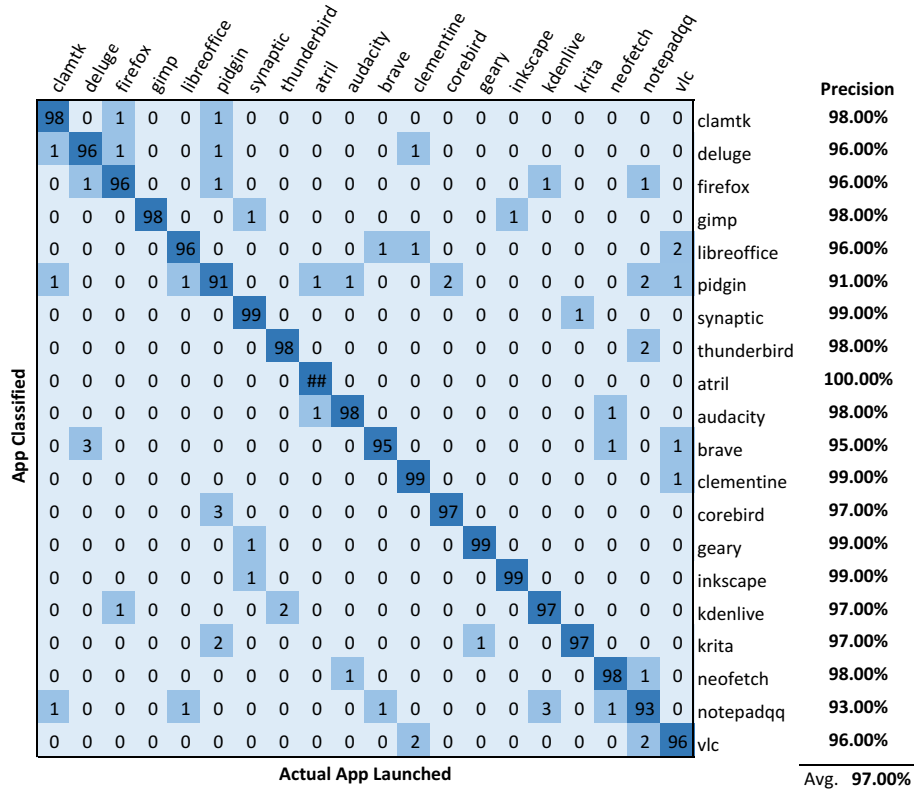


Figure 5.8: Heatmap for classification results of inferring the launched app on Ubuntu

We also use these two classifiers to evaluate how the time period affects the precision for inferring the activities. The results are presented in Figure 5.10. As presented in this figure, we can find that the smallest time period of sampling for

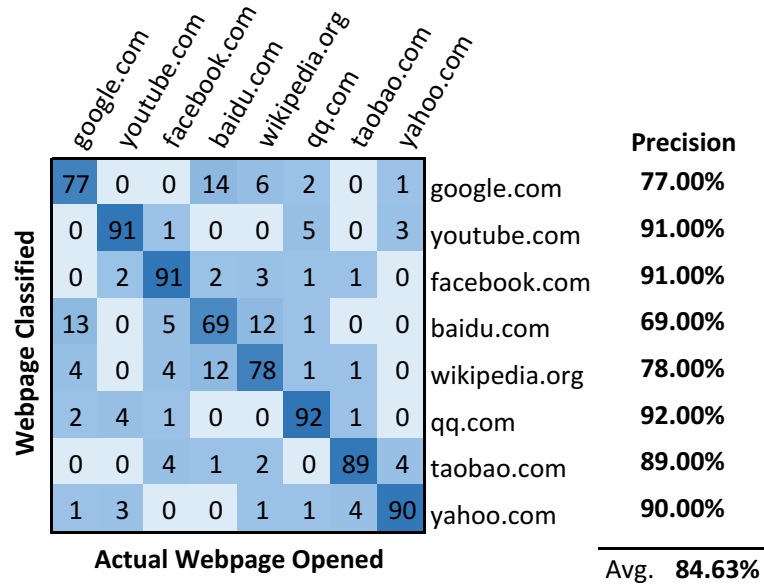


Figure 5.9: Heatmap for classification results of inferring the loaded webpage on Ubuntu

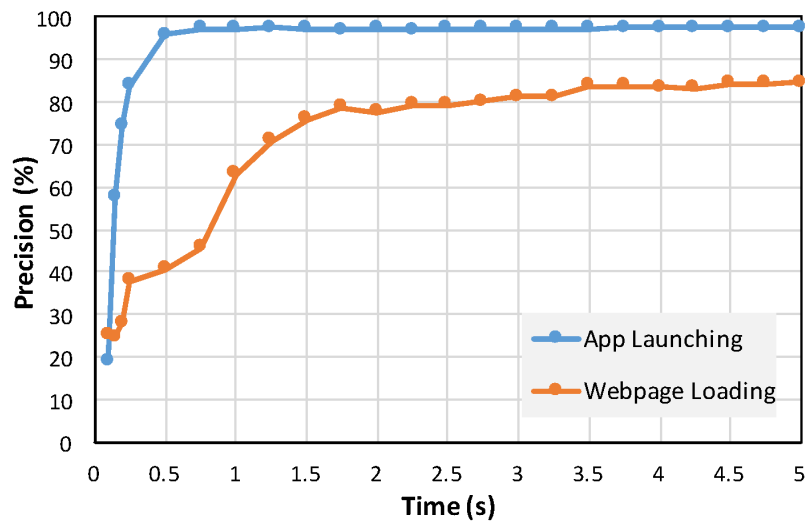


Figure 5.10: The precision of inferring app activities using different lengths of interrupt timing series on Ubuntu

reaching 95% precision of inferring the launched app is 0.5 seconds, while reaching about 80% precision for inferring webpage loading is 1.5 second. This difference implies that launching applications splashing differently from the very beginning while loading webpages with the same browser splashes differently after the browser is opened.

There are several main differences of experimental results on Android and Ubuntu. First of all, app identifying precision on Ubuntu is higher than Android. On Android, the best precision is 92.70% while 97.65% on Ubuntu. The reasons are that Android, as less powerful devices, is weak in handling high rate sampling which may bring in noises and GPU interrupts on Android may not provide as much information as desktop. Second, the precision webpage loading identification on Android is higher than Ubuntu. This is because the relative information contains in the content of frames displayed is less on Ubuntu.

5.5.3 Noises Analysis

Different from the mobile platform, GUIs at multiple applications are usually displayed simultaneously on the screen of a Linux system. For example, users may open more than one application at the same time and the layouts of different apps may overlap. In order to find out the strength of the classification model for resisting noises from background applications, we conduct an experiment in which the target application is launched while there are background applications running with visible windows on the desktop. The first scenario corresponds to a webpage is opened and the launched app will overlap the layout of the background webpage. Another scenario is a video playing in the background while the app is launched. With IB1 classifier, the precision of identifying the launched app with a webpage in the background is 66.00%. The precision of identifying launched app with a video playing in the background is lower than 10%. The results show that static background app has smaller effects on the precision of the model, while dynamic background app

will disable the attacks.

5.6 Discussion

In this section, we discuss the strength and weakness points of the proposed side-channel attack.

The proposed side-channel attacks can effectively disclose app activities on Linux and Linux-based systems without access large amount of information. This is important as it is more and more difficult to access other processes' information, especially on Android. By analyzing the launched app and the activities within apps, we can detect the user behavior on the devices. The attacks have low overheads. We don't need to modify the system, obtain root permission or hook the target applications. We only need to start a small logging program on the target devices without any permission. This is feasible with phishing or repacking techniques and difficult to be detected.

However, the proposed attacks have several weaknesses. First, it is not feasible on devices which do not allow third-party apps to access `/proc/interrupts` file. However, there are still a considerable amount of systems provide accesses to this file. Second, the attack is sensitive to noises from background applications which hardly use GPU and generates interrupts. In the future, we would like to investigate how to reduce noises generated by background applications. Third, the built models are specific for different platforms and scenarios. The GPU interrupt mechanisms are different for different types of GPU and drivers. Hence, the attacks need a training procedure to build models for the target platform.

5.7 Summary

In this chapter, we aim to use GPU interrupt timing information to infer app activities on Linux and Linux-based systems. We use the timing series of GPU interrupt

to feature the app activities and classify the features to identify the activities with machine learning techniques. With this method, we can effectively identify app activities. It can be used in both inferring the launched app and the activities in an application. We conduct experiments on Android and Ubuntu. The results show that the identification models have high precision and are effective with static background noises.

Chapter 6

Application of Dynamic Analysis in a Malware-Spreading Study

This chapter demonstrates that dynamic analysis under non-rooted running environment enables analysis of malware spreading. This chapter proposes to simulate a malware-spreading environment under real running environment and analyze different factors that would affect the malware spreading.

6.1 Introduction

Smartphone has gained tremendous adoption and becomes more and more necessary in our daily lives since its inception in 1973. According to GSMA real-time intelligence data, there are now over 8.98 Billion mobile connections worldwide, which surpasses the current world population of 7.69 Billion implied by UN digital analyst estimates [8]. Unfortunately, the popularity of smartphone makes it one of the most popular platforms targeted by attackers. Malware is one of the most serious threats on mobile platforms, especially on Android, the most popular mobile system (with more than 85% market sharing [38]).

Malware authors aggressively add different features to upgrade the severity of the malware. For example, they may use dynamic code loading, Java-reflection,

encryption, and obfuscation to avoid detection by dynamic analysis [47]. A new feature that shows up on modern malware recently is that it may try to infect devices by spreading through mobile networks (e.g., SMS or social networks). The propagation of malware can scale up the attacks to get millions of people being infected. Normally, a malware will access the contact on the device and send malicious content (e.g., malicious links or videos) to the contacts. If the contacts receive and click the malicious content, the malware will automatically be installed on their devices to conduct malicious behavior, such as stealing private information.

To spread out, phishing has been one of the most popular spreading methods adopted by Android malware. As reported by authorities, devices can easily get infected when users click malicious links in SMS, instant messages or emails:

1. “Android Banking Trojan MoqHao Spreading via SMS Phishing in South Korea” McAfee Aug, 2017
2. “Android Mazar malware that can wipe phones spread via SMS” BBC Feb, 2016
3. “A clever WhatsApp scam could trick you into spreading a virus to friend” buzz.ie May, 2017

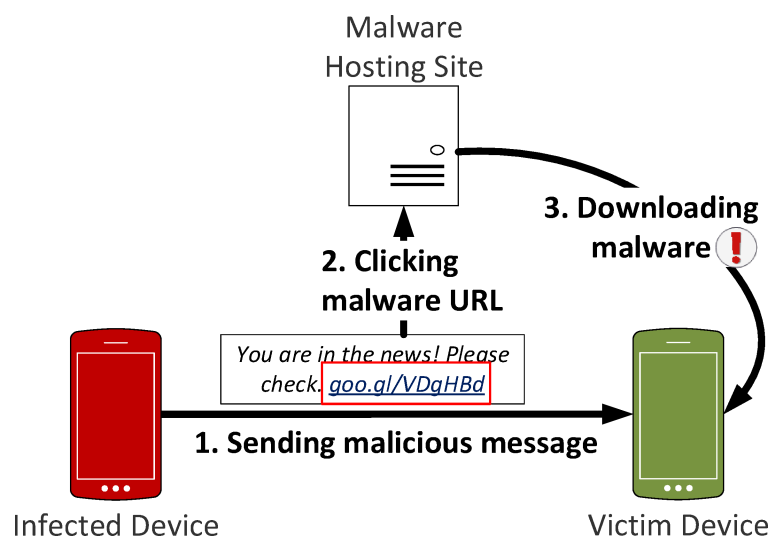


Figure 6.1: Malware spreading among mobile devices.

Figure 6.1 presents how malware spreads among different devices. Normally, an infected device would try to send malicious messages to the victim device. Malware is able to get the address of the victim devices by scanning the contact list of the mobile user. It typically sends messages exhaustively to any address it can obtain. The spreading message will contain malicious links or media files. By clicking the links or downloading the files, the victim device will automatically download malware from the hosting site and get infected.

To analyze how different factors would affect the spreading of malware, we propose a simulation study for analyzing malware spreading among mobile users. The proposed dynamic analysis techniques work on non-rooted devices in the wild which require deployment on real users devices. This study demonstrates the feasibility of applying dynamic analysis to real users devices for analyzing. We dynamically monitor different factors (e.g., time and location) when the message spreads out and record the recipient's response on the message. To demonstrate the effectiveness of dynamic analysis on crowd-sourcing, we conduct a user study with real users on the simulation system we build. We recruit 48 participants to spread out seemingly malicious messages to their contacts. The seemingly malicious message contains a URL to simulate the malicious malware spreading messages. However, the webpage that the URL leads the user to is totally harmless. With a provided app, participants can customize the factors about the messages (e.g., the method for sending the message, the time when the messages are sent out, and the content of the message). By analyzing the messages sent out by the participants and the responses from the receiver, we emphasize the factors that would affect malware spreading with instant messaging.

6.2 Problem Statement

In this study, we hope to identify under which circumstance a victim would be more likely to get infected by a malicious message. We conduct a user study to simulate

the malware spreading via a seemingly malicious message that contains a phishing link but without any harmless content. Participants in the user study are required to send seemingly malicious messages to their contacts. The link leads the user to a webpage for recording users' reactions without any harmful content. We will record whether the link is clicked by the contact of the participant to represent whether the malware successfully spreads.

We hope to use this study to analyze how different factors would affect the malware spreading, e.g., whether using WhatsApp will reach higher spreading rate compare to other messaging tools. We take several factors that would affect the spreading into consideration. While the malware spreads among mobile users, it usually steals the users' contact lists and sends messages to them. From the aspect of the recipient, the relationship and interaction frequency with the sender determine their trust degree on the received messages. If the messages are sent by a familiar person, they may start to read the content of the messages. Otherwise, there is a high possibility that the recipient may ignore the message. The other factors determine whether the user would be suspicious about the message. If the messages are received in abnormal time via a channel rarely used for their conversation, they would be more suspicious on it. The content of the message also matters since they would chat more frequently on specific topics. An abnormal slang may also raise the recipient's suspicion. We also consider the location of the recipient when they click the URL to see whether they would be more suspicious about the messages in specific locations. For example, if they are at the same location, the recipient may ask about the message to avoid infection. We choose these factors and classify each of them as follows:

1. Relationship between message sender and receiver: Friend / Family Member / Acquaintance / Stranger / Colleague
2. Frequency of interaction between sender and receiver: Daily / Weekly / Monthly / Rarely

3. Time of sending the messages: Weekday / Weekend
4. Method of the spreading: SMS / Facebook / WeChat / WhatsApp / Gmail
5. Wording of the message: Personalized / Impersonalized
6. Location of the receiver

6.3 Simulation System

In this section, we present the design of the system we build for simulating malware spreading among multiple users.

6.3.1 Overview

Figure 6.2 presents the overview of the system. The server is a secure server in SMU for managing users and different resources. The mobile client is a third-party application we developed to deploy monitoring and message customization on the users' devices. During runtime, the mobile client helps users to send out seemingly malicious messages to their contacts. The responses from recipients of the messages will be recorded by our server. We also conduct a post-survey on the recipients to obtain their feedback on the study. The feedback is also stored on the server.

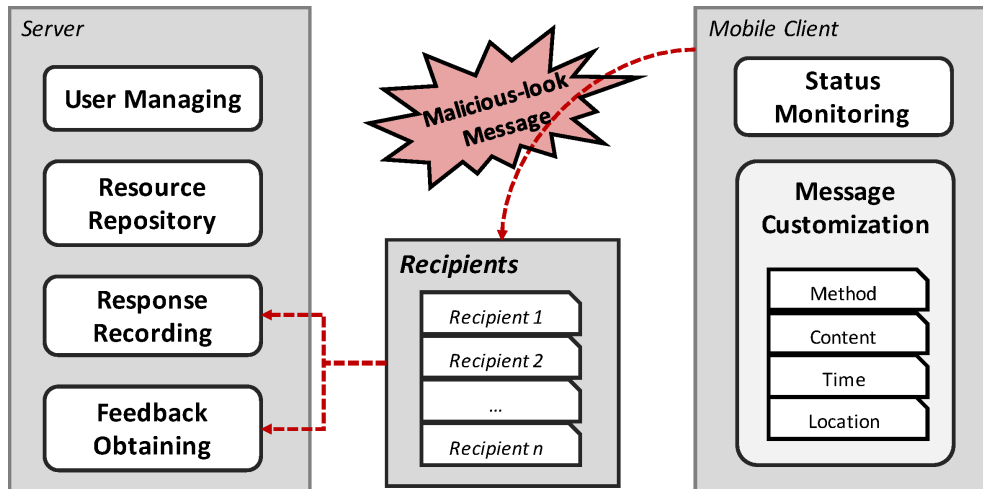


Figure 6.2: Overview of the system for simulation of malware spreading

6.3.2 Seemingly malicious Message

The main goal of this system is to help users send out seemingly malicious messages and record responses from the recipients. We present the constitution of the seemingly malicious messages and how to send it out with our mobile client in this section.

To simulate the malware spreading messages, the message sent out from the client contains two parts - description of offers (e.g., coupon of restaurants) and a corresponding URL link. A sample message as follows:

Hi, I am planning for Phuket, do you wanna join? Just checked flights are really cheap: www.singaporeblogzone.com/blog?s_id=8&r_id=2

In the resource repository on the server, we have ten sample messages for the users to choose from. These messages contain different topics of offers, such as anti-virus tools and flight deals. The user can also customize the message to include his own slang. The URL link leads the users to a website managed by our server. We embed id of both the sender and the recipient in the link to identify them when the link is clicked.

Figure 6.3 presents the steps for users to customize the seemingly malicious messages after choosing the recipients. First of all, the users should choose the main topic of messages from the ten templates stored on the server. Then, they should decide whether to customize the message to include their own slang. Finally, they need to choose the methods for sending out messages. We include WhatsApp, Gmail, WeChat and Facebook messages as the spreading methods. SMS is an additional method in case the recipient does not use WhatsApp.

6.3.3 Status Monitoring

In order to find out the factors that would affect the success rate of malware spreading, we monitor the status of the users' devices when the message is sent out. Mainly, we consider five factors on the users' devices. The first factor is the re-

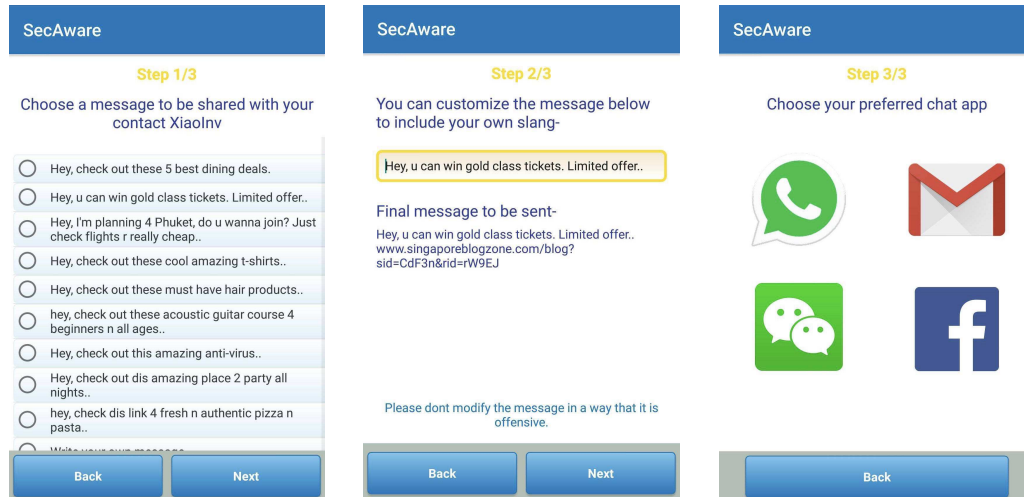


Figure 6.3: Steps for users to customize the messages to be sent out

relationship between the sender and the recipient. We also evaluate their familiarity with contact frequencies, e.g., contact every day or once a week. Before sending out messages, the sender needs to choose their familiarity with the recipient. The second factor is the time when the message is sent out. The user can choose when to send out the messages and the client app will record it. The third one is the method of sending messages. We provide different choices of the spreading methods for users and record their selection at runtime. We include popular social network messaging tools, such as Gmail, Facebook, WhatsApp, and WeChat. The fourth one is the topic of the message and whether it is customized by the user. The last one is the location of the recipient when the message is clicked. We record the IP addresses of the recipients' devices when they click the links.

6.4 Simulation Study

With the proposed simulation system, we conduct a user study to simulate malware spreading and record user reactions to the malicious messages. This study has been approved by SMU IRB under approval number IRB-17-109-A135(1217). With the approval, we have conducted user study among students and staffs in SMU and their contacts.

6.4.1 Recruitment

In this study, we recruit two groups of participants from Singapore. The first group (Group A) of participants are those who send out the malware spreading message. The other group (Group B) of participants are the message recipients. We recruit students and staffs from Singapore Management University (SMU) as Group A participants. Group B participants are recruited through invitation links sent by Group A since only Group A can reach their contacts and send messages to them following the data privacy protection policies.

The recruitment is conducted via both online and physical advertisement to SMU community, including emails, printed posters, and so on. The advertisement will explain the goal, procedure, and compensation to the potential participants. The one who is interested in this study can scan the QR code in the advertisements to download our app named SecAware and register the study with it. In the informed consent, participants in Group A are aware of the whole story of this study. After successful registration, they can send invitation links to their contacts to recruit Group B participants. The invitation link will lead the potential participants to a webpage which explains that they would receive several messages. However, we don't provide the information about the whole study to them, since we need to record their real actions to the messages.

The study started from June 1 2019 and lasted for five weeks. In the first two weeks, we recruit participants and encourage them to invite their contacts to join the study. In the third and fourth weeks, participants start to send messages to their contacts and research team will also send the messages. The last week, participants need to finish the post-survey. After that, we will pay them according to their contribution to this study. They would receive cash, Starbucks e-gift card, and chances of winning an iPhone XR as compensation. The lucky draw will be held in August 2019. Only with Group A's contacts successfully registered the study as Group B, both groups of participants can get compensation.

6.4.2 Seemingly malicious Message

The message to be sent to Group B during the user study (not the invitation message) contains two parts - description of offers (e.g., discount coupon of restaurants) and a corresponding URL link. The only information embedded in this link is the ID of Group A and ID of Group B. These IDs are numeric, assigned by our app and do not contain any personal information.

Group A can then modify the message (except the link) to make it personalized to the specific contact recipient (i.e. Group A can edit the message to include his own slang which he generally uses while having a conversation with different contacts). However, we will provide topics to Group A based on which he can frame his own message. The topics are the same as the deals on the webpage that the URL points to (contents of which are described in the next paragraph). For example, a sample message is: *Hey, check out these 5 best dining deals. <https://...>* On the webpage, there would be similar deal ads. The research team will communicate to Group A that Group A should not modify the message in a way that will seem offensive to Group B and generally the message should be in good taste.

6.4.3 Spreading

Group A can start to send messages to Group B after Group B registers the study for one week. We choose one week as the period to refresh Group B's memory according to the Ebbinghaus curve. Ebbinghaus curve shows that memory retention is 25% after one week. Group A will first choose a participant in Group B invited by him/her and specify their relationship and contact frequency. After this, Group A can customize the message and start to send the message out.

In order to find out the difference between sending the spreading message by acquaintance or strangers, we send the message twice to the same participant in Group B by both the Group A participant and the research team (unknown number to Group B). Before sending out messages, group A can choose to send the message

first or let the research team send the message first. After one week, the other entity will send the same message again to the chosen Group B participant.

6.4.4 Results

We have implemented the dynamic analysis system including the app for group A to customize and send messages and the server application for logging reactions in group B. In total, we have recruited 48 participants from SMU as Group A, 90 of their contacts registered the study as Group B. During the study, 144 messages have been sent to 84 participants in Group B. Among these messages, 91 are sent by participants in Group A and 53 are sent by the research team. If the recipient clicks the seemingly malicious URL in the message, we call this recipient “infected”. In total, 22 participants in Group B get “infected” in this simulation of malware spreading. We analyze details of the messages and clicks on the links in the seemingly malicious messages and present the results as follows.

Figure 6.4 presents the times of clicks on the seemingly malicious URL and the corresponding numbers of contacts (e.g., eight participants clicked the URL for one time). Some of the “infected” participants click the link in the seemingly malicious message for once. However, there are a considerable number of participants who would click the link. From the survey on the participants, they are deceived by the messages and would like to figure out the content on the webpage, so they click the links for many times to see whether there is something wrong on the network or webpage. This presents that participants who click the link for multiple times would be vulnerable to the phishing messages.

First, we analyze the infection rate with different message sending methods which are a simulation of the method of malware spreading. Figure 6.5 presents the number of infected participants from different spreading methods, including Facebook, Gmail, SMS, WhatsApp, and WeChat. During this study, we do not have samples of message sending via Facebook. Among the other methods, we found

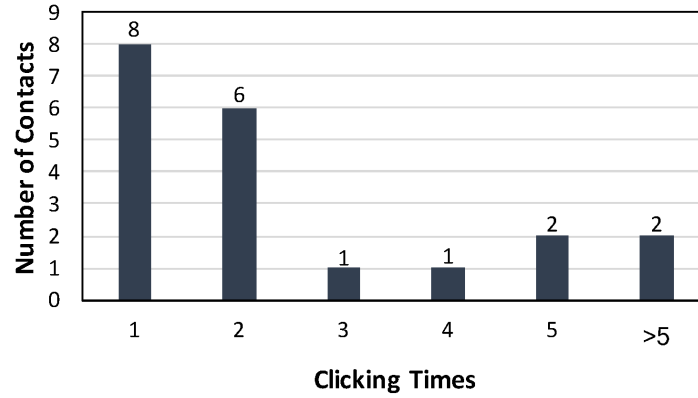


Figure 6.4: The number of participants who clicked the URL in the seemingly malicious message for different times

that WeChat and WhatsApp have a higher infection rate during malware spreading. It seems people are more vigilant to email and SMS. This may due to the popularity of phishing attacks via email and SMS.

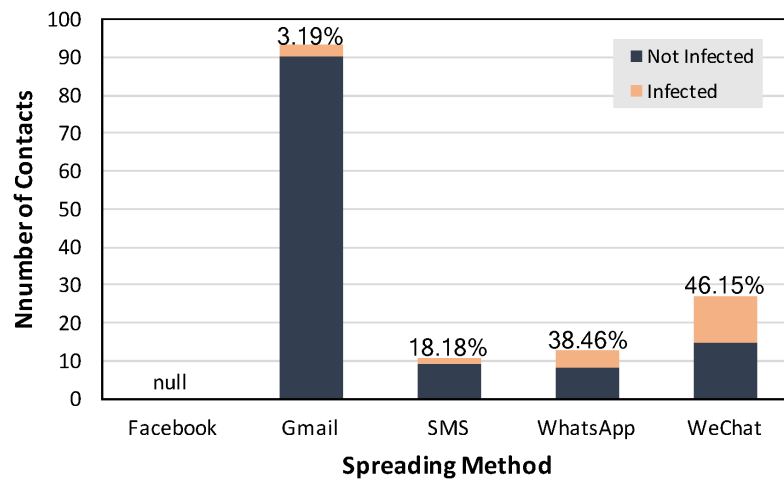


Figure 6.5: The number of infected participants from different spreading methods

Second, we analyze the infection rate of participants who have a different relationship with the message sender. In order to find out whether people would be infected by messages sent by strangers, we also send messages to the participants from the research team. Figure 6.6 presents how relationship between the sender and the recipient would affect the infection rate. From this figure, we can find that participants with family, friend and colleague relationships are more likely to be infected. They are more vigilant to messages sent by their acquaintances or strangers.

Sending the messages by research team shows whether the participants would get educated in this study. Among the messages to the 84 participants in Group B, only 1 get infected for the second time. This shows that lots of participants would get educated in this study, which means few people would get infected by the malware spreading messages for twice. We also analyze the contact frequency between the sender and recipient. In our dataset, more than 85% are marked as contacted every day by Group A, and only seven are labeled as frequently contacted. A possible reason is that some pairs of participants have close and reliable relationships but may not contact via messages frequently. This often happens among colleagues, families, and close friends.

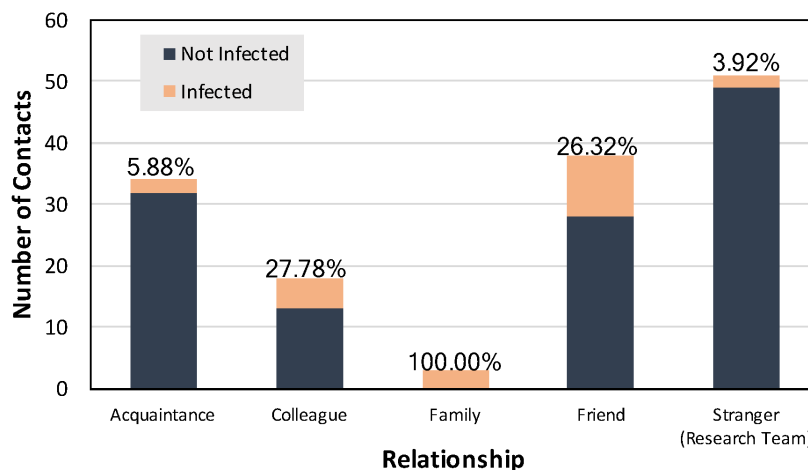


Figure 6.6: The number of infected participants who have a different relationship with the sender

Third, we analyze the number of clicks on the seemingly malicious URL in the messages received by the participants in different time periods as presented in Figure 6.7. From this figure, participants react to the messages more actively at noon and in the afternoon. They are more likely to be infected at noon and in the afternoon. The first possible reason is that people may be not that active in reading and replying messages in the morning and at night. Another reason is that the participants' vigilance falls at noon and in the afternoon. For estimated duration between the message receiving and the URL clicking, it takes shorter than 5 mins for nine participants, shorter than 2 hours for eleven, and longer than 5 hours for the

other two. This means most of the participants would get infected by the message in 2 hours, otherwise, they would ignore the message.

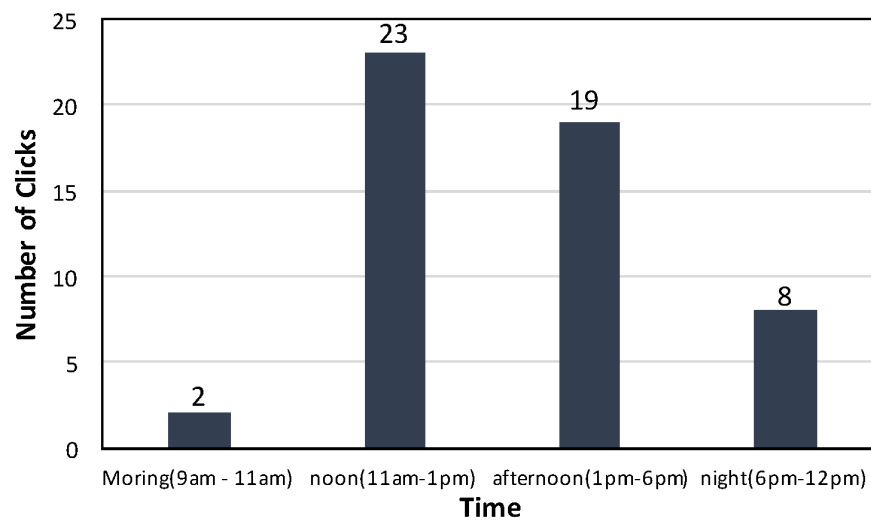


Figure 6.7: The number of participants who are infected in different time

We have also recorded the IP addresses of the infected participants for identifying whether location would affect the infection rate. Due to the inaccuracy of location inferring via IP address, we can only get a very general address (e.g., central of Singapore) of the participant who clicks the seemingly malicious URL. As the server is also located in SMU, we can also identify those who get infected are using SMU network or not. In total, we have recorded 30 distinct IP addresses while the URLs are clicked. Seven IP addresses are from SMU; the others are from different part of Singapore. This implies that people are more likely to be infected when they are using their personal network access rather than network from their school or company. Among the messages sent out, only 3 are sent from SMU campus and none of them gets “infected” participants in Group B. For the messages sent from other places, it is difficult to identify the location. Hence, the current data is not sufficient to provide clues about whether the pair of participants are in the same location would affect the infection rate or not.

Among the 144 messages sent out, 42 of them are customized by the sender. And, 10 (out of 22) of the clicks are on the URL in the customized messages. The

infection rate of customized messages is 23.80% and the infection rate of the default messages is 15.71%. Customized messages lead to slightly higher infection rate. This shows the participants are more vulnerable to customized messages which are more likely to be edited by their contacts.

In summary, we would have the following results corresponding to the questions raised in Section 6.2:

1. More reliable relationship between message sender and receiver (family and colleague) would increase the success rate of the spreading.
2. Higher frequency of interaction between sender and receiver corresponds to a higher success rate.
3. During noon and afternoon, people may be more likely to be infected by malware spreading messages.
4. WeChat, WhatsApp and SMS are the methods which lead to a higher success rate in the spreading. From the post-survey conducted by the receivers, similar messages would be more likely to be found in WhatsApp messages.
5. Customized messages which are personalized by the sender leads to a higher success rate in malware spreading.

6.5 Summary

In this chapter, we aim to analyze how external environment and human-related factors affect malware spreading. We simulate a popular malware spreading method with which malware spreads among mobile users via instant messages. We conduct a user study to simulate the spreading and collect the data to record the external factors as well as users' responses to the malware spreading messages. We find that spreading method, relationship, and contact frequency significantly affect the spreading of malware via instant messages.

Chapter 7

Conclusion

App analysis on non-rooted devices in the wild is a more and more important research problem, since security mechanisms are introduced by Android to prevent accessing other apps' runtime information. In this thesis, we address the importance of this problem and propose solutions of dynamic analysis on non-rooted devices without system modification or app instrumentation.

We first present a code-reuse-based technique for protecting Android applications which enhances the concealment of both Java and native code in Android apps through hiding essential code. Our evaluation shows that the limited binary resources in Android apps are sufficient for applying code-reuse-based obfuscations. We further implement AndroidCubo semi-automate the process of obfuscating essential code. Examples present that it is practical to protect applications with AndroidCubo.

Then, we present our efforts on the dynamic analysis of app behavior under unmodified and non-rooted devices. We propose UpDroid - a system for dynamically monitoring Android apps' sensitive behavior. It uses different APIs to monitor Android system at runtime and leverages learning to rank technique to identify the initiator of the detected behavior. We use the permission coverage, the runtime experiments and the comparison with the traditional API hooking method to demonstrate the capabilities of UpDroid. The results show that UpDroid can de-

tect sensitive behavior that manipulates the resources of the devices and identify the apps that trigger the behavior.

We also propose to use GPU interrupt timing information to infer app activities on Linux and Linux-based systems. We use the timing series of GPU interrupt to feature the app activities and classify the features to identify the activities with machine learning techniques. With this method, we can effectively identify app activities. It can be used in both inferring the launched app and the activities in an application. We conduct experiments on Android and Ubuntu. The results show that the identification models have high precision and are effective with static background noises.

Finally, we analyze how external environment and human-related factors affect malware spreading. We simulate a popular malware spreading method with which malware spreads among mobile users via instant messages. We conduct a user study to simulate the spreading and collect the data to record the external factors as well as users' responses to the malware spreading messages.

Bibliography

- [1] Dexguard. <https://www.guardsquare.com/dexguard>.
- [2] Dexguard. <https://dexprotector.com/>.
- [3] B. Alll and C. Tumbleson. Dex2jar: Tools to work with android. dex and java. class files.
- [4] S. Anand, M. Naik, M. J. Harrold, and H. Yang. Automated concolic testing of smart-phone apps. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE '12)*, page 59. ACM, 2012.
- [5] AppBrain. Top android sdk versions, 2018.
- [6] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie. Pscout: analyzing the android permission specification. In *Proceedings of the 2012 ACM conference on Computer and communications security (CCS '12)*, pages 217–228. ACM, 2012.
- [7] M. Backes, S. Bugiel, E. Derr, P. McDaniel, D. Ocateau, and S. Weisgerber. On demys-tifying the android application framework: Re-visiting android permission specifica-tion analysis. In *Proceedings of 25th USENIX Security Symposium (USENIX Security '16)*, pages 1101–1118. USENIX Association, 2016.
- [8] bankmycell. How many people have phones in the world? <https://www.bankmycell.com/>.
- [9] A. Bianchi, Y. Fratantonio, C. Kruegel, and G. Vigna. Njas: Sandboxing unmodified applications in non-rooted devices running stock android. In *Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM '15)*, pages 27–38. ACM, 2015.
- [10] T. Bläsing, L. Batyuk, A.-D. Schmidt, S. A. Camtepe, and S. Albayrak. An android application sandbox system for suspicious software detection. In *Proceedings of the 5th International Conference on Malicious and Unwanted Software (MALWARE '10)*, pages 55–62. IEEE, 2010.
- [11] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang. Jump-oriented programming: a new class of code-reuse attack. In *Proc. ACM ASIACCS*, 2011.
- [12] D. Bogdanas. Dperm: Assisting the migration of android apps to runtime permissions. *arXiv preprint arXiv:1706.05042*, 2017.
- [13] E. Buchanan, R. Roemer, H. Shacham, and S. Savage. When good instructions go bad: generalizing return-oriented programming to risc. In *Proc. ACM CCS*, 2008.

- [14] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani. Crowddroid: behavior-based malware detection system for android. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices (SPSM '11)*, pages 15–26. ACM, 2011.
- [15] Z. Cao, T. Qin, T.-Y. Liu, M.-F. Tsai, and H. Li. Learning to rank: from pairwise approach to listwise approach. In *Proceedings of the 24th international conference on Machine learning (ICML '07)*, pages 129–136. ACM, 2007.
- [16] N. Carlini and D. Wagner. Rop is still dangerous: Breaking modern defenses. In *USENIX Security Symposium*, 2014.
- [17] J.-T. Chan and W. Yang. Advanced obfuscation techniques for java bytecode. *Journal of Systems and Software*, 71(1):1–10, 2004.
- [18] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy. Return-oriented programming without returns. In *Proc. ACM CCS*, 2010.
- [19] Q. A. Chen, Z. Qian, and Z. M. Mao. Peeking into your app without actually seeing it: {UI} state inference and novel android attacks. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, pages 1037–1052, 2014.
- [20] S. R. Choudhary, A. Gorla, and A. Orso. Automated test input generation for android: Are we there yet? In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE '15)*, pages 429–440. IEEE, 2015.
- [21] C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. Technical report, Department of Computer Science, The University of Auckland, New Zealand, 1997.
- [22] C. Collberg, C. Thomborson, and D. Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 184–196. ACM, 1998.
- [23] V. Dang. Ranklib. 2013.
- [24] S. K. Dash, G. Suarez-Tangil, S. Khan, K. Tam, M. Ahmadi, J. Kinder, and L. Cavallaro. Droidscribe: Classifying android malware based on runtime behavior. *Mobile Security Technologies (MoST 2016)*, 7148:1–12, 2016.
- [25] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy. Return-oriented programming without returns on arm. *System Security Lab-Ruhr University Bochum, Tech. Rep*, 2010.
- [26] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy. Privilege escalation attacks on android. In *Information Security*. Springer, 2011.
- [27] L. Davi, D. Lehmann, A.-R. Sadeghi, and F. Monrose. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *USENIX Security Symposium*, 2014.
- [28] W. Diao, X. Liu, Z. Li, and K. Zhang. No pardon for the interruption: New inference attacks on android through interrupt timing analysis. In *Proceedings of the 2016 IEEE Symposium on Security and Privacy (S&P '16)*, pages 414–432. IEEE, 2016.
- [29] E. Dupuy. Jd-gui: Yet another fast java decompiler. URL <http://java.decompiler.free.fr/?q=jdgui>/accessed March, 2012.

- [30] fgwei. Android api to permission mapping extractor, 2017.
- [31] E. Göktaş, E. Athanasopoulos, M. Polychronakis, H. Bos, and G. Portokalidis. Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard. In *23rd USENIX Security Symposium, San Diego, CA*, pages 417–432, 2014.
- [32] Google. Jni tips. <http://developer.android.com/>.
- [33] Google. Application security, 2017.
- [34] Google. Fileobserver, 2017.
- [35] Google. Ui/application exerciser monkey, 2017.
- [36] G. Holmes, A. Donkin, and I. H. Witten. Weka: A machine learning workbench. 1994.
- [37] H. Huang, K. Chen, C. Ren, P. Liu, S. Zhu, and D. Wu. Towards discovering and understanding unexpected hazards in tailoring antivirus software for android. In *Proceedings of the 10th ACM SIGSAC Symposium on Information, Computer and Communications Security (AsiaCCS '15)*, pages 7–18. ACM, 2015.
- [38] I. D. C. (IDC). Smartphone market share, 2018.
- [39] Y. Jing, Z. Zhao, G.-J. Ahn, and H. Hu. Morpheus: automatically generating heuristics to detect android emulators. In *Proceedings of the 30th Annual Computer Security Applications Conference (ACSAC '14)*, pages 216–225. ACM, 2014.
- [40] M. Karami, M. Elsabagh, P. Najafiborazjani, and A. Stavrou. Behavioral analysis of android applications using automated instrumentation. In *Proceedings of the 2013 IEEE 7th International Conference on Software Security and Reliability-Companion (SERE-C '13)*, pages 182–187. IEEE, 2013.
- [41] M. Koscielnicki. Nvidia hardware documentation, 2019.
- [42] E. Lafortune et al. Proguard. <http://proguard.sourceforge.net>.
- [43] P. Lantz, A. Desnos, and K. Yang. Droidbox: Android application sandbox, 2017.
- [44] K. Lu, S. Xiong, and D. Gao. Ropsteg: program steganography with return oriented programming. In *Proc. ACM CODASPY*, 2014.
- [45] H. Ma, K. Lu, X. Ma, H. Zhang, C. Jia, and D. Gao. Software watermarking using return-oriented programming. 2015.
- [46] A. Machiry, R. Tahiliani, and M. Naik. Dynodroid: An input generation system for android apps. In *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE '13)*, pages 224–234. ACM, 2013.
- [47] A. Mahboubi, S. Camtepe, and H. Morarji. A study on formal methods to generalize heterogeneous mobile malware propagation and their impacts. *IEEE Access*, 5:27740–27756, 2017.
- [48] S. Marchesin. Linux graphics drivers: an introduction. 2012.
- [49] MindMac. Android eagleeye, 2016.
- [50] T. Morkel, J. H. Eloff, and M. S. Olivier. An overview of image steganography. In *ISSA*, 2005.

- [51] S. Mutti, Y. Fratantonio, A. Bianchi, L. Invernizzi, J. Corbetta, D. Kirat, C. Kruegel, and G. Vigna. Baredroid: Large-scale analysis of android apps on real devices. In *Proceedings of the 31th Annual Computer Security Applications Conference (ACSAC '15)*, pages 71–80. ACM, 2015.
- [52] Nergal. The advanced return-into-lib(c) exploits (pax case study). *Phrack magazine*, 4(58), 1996.
- [53] Z. Ning and F. Zhang. Ninja: Towards transparent tracing and debugging on arm. In *Proceedings of 26th USENIX Security Symposium (USENIX Security '17)*, pages 33–49. USENIX Association, 2017.
- [54] T. Petsas, G. Voyatzis, E. Athanasopoulos, M. Polychronakis, and S. Ioannidis. Rage against the virtual machine: hindering dynamic analysis of android malware. In *Proceedings of the 7th European Workshop on System Security (EuroSec '14)*, pages 5:1–5:6. ACM, 2014.
- [55] Z. Qian, Z. M. Mao, and Y. Xie. Collaborative tcp sequence number inference attack: how to crack sequence number under a second. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 593–604. ACM, 2012.
- [56] B. Rashidi, C. Fung, A. Nguyen, T. Vu, and E. Bertino. Android user privacy preserving through crowdsourcing. *IEEE Transactions on Information Forensics and Security*, 13(3):773–787, 2017.
- [57] J. Salwan and A. Wirth. Ropgadget, 2012.
- [58] E. J. Schwartz, T. Avgerinos, and D. Brumley. Q: Exploit hardening made easy. In *USENIX Security Symposium*, pages 25–41, 2011.
- [59] D. Seal. *ARM architecture reference manual*. Pearson Education, 2001.
- [60] A. Shabtai, U. Kanonov, Y. Elovici, C. Glezer, and Y. Weiss. ”andromaly”: a behavioral malware detection framework for android devices. *Journal of Intelligent Information Systems*, 38(1):161–190, 2012.
- [61] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proc. ACM CCS*, 2007.
- [62] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *Proc. IEEE Symposium on Security and Privacy*. IEEE, 2013.
- [63] M. Spreitzenbarth, F. Freiling, F. Ehtler, T. Schreck, and J. Hoffmann. Mobile-sandbox: having a deeper look into android applications. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing (SAC '13)*, pages 1808–1815. ACM, 2013.
- [64] K. Tam, S. J. Khan, A. Fattori, and L. Cavallaro. Copperdroid: Automatic reconstruction of android malware behaviors. In *Proceedings of the 22nd Annual Network and Distributed System Security Symposium (NDSS '15)*, 2015.
- [65] M. Tran, M. Etheridge, T. Bletsch, X. Jiang, V. Freeh, and P. Ning. On the expressiveness of return-into-libc attacks. In *Recent Advances in Intrusion Detection*, pages 121–141. Springer, 2011.

- [66] T. Vidas and N. Christin. Evading android runtime analysis via sandbox detection. In *Proceedings of the 9th ACM SIGSAC Symposium on Information, Computer and Communications Security (AsiaCCS '14)*, pages 447–458. ACM, 2014.
- [67] D. Y. Wang, S. Savage, and G. M. Voelker. Cloak and dagger: dynamics of web search cloaking. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 477–490. ACM, 2011.
- [68] T. Wang, K. Lu, L. Lu, S. Chung, and W. Lee. Jekyll on ios: When benign apps become evil. In *Proc. Usenix Security*, 2013.
- [69] X. Wang, S. Zhu, D. Zhou, and Y. Yang. Droid-antirm: Taming control flow anti-analysis to support automated dynamic analysis of android malware. In *Proceedings of the 33rd Annual Computer Security Applications Conference (ACSAC '17)*, pages 350–361. ACM, 2017.
- [70] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *Proc. ACM CCS*, 2012.
- [71] F. Wei, Y. Li, S. Roy, X. Ou, and W. Zhou. Deep ground truth analysis of current android malware. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA '17)*, pages 252–276. Springer, 2017.
- [72] R. Winsniewski. Apktool: A tool for reverse engineering android apk files. <http://ibotpeaches.github.io/Apktool/>.
- [73] M. Y. Wong and D. Lie. Intellidroid: A targeted input generator for the dynamic analysis of android malware. In *Proceedings of the 23rd Annual Network and Distributed System Security Symposium (NDSS '16)*, 2016.
- [74] D. Wu, R. K. C. Chang, W. Li, E. K. T. Cheng, and D. Gao. Mopeye: Opportunistic monitoring of per-app mobile network performance. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 445–457, Santa Clara, CA, 2017. USENIX Association.
- [75] D.-J. Wu, C.-H. Mao, T.-E. Wei, H.-M. Lee, and K.-P. Wu. Droidmat: Android malware detection through manifest and api calls tracing. In *Information Security (Asia JCIS), 2012 Seventh Asia Joint Conference on*, pages 62–69. IEEE, 2012.
- [76] Xposed. Welcome to the xposed module repository!, 2017.
- [77] L.-K. Yan and H. Yin. Droidscape: Seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. In *Proceedings of 21th USENIX Security Symposium (USENIX Security '12)*, pages 569–584. USENIX Association, 2012.
- [78] C. Yang, Z. Xu, G. Gu, V. Yegneswaran, and P. Porras. Droidminer: Automated mining and characterization of fine-grained malicious behaviors in android applications. In *Proceedings of the 2014 European symposium on research in computer security (ESORICS '14)*, pages 163–182. Springer, 2014.
- [79] L. Yang, N. Boushehrinejadmoradi, P. Roy, V. Ganapathy, and L. Iftode. Short paper: enhancing users' comprehension of android permissions. In *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices*, pages 21–26. ACM, 2012.

- [80] K. Zhang and X. Wang. Peeping tom in the neighborhood: Keystroke eavesdropping on multi-user systems. In *USENIX Security Symposium*, volume 20, page 23, 2009.
- [81] N. Zhang, K. Yuan, M. Naveed, X. Zhou, and X. Wang. Leave me alone: App-level protection against runtime information gathering on android. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy (S&P '15)*, pages 915–930. IEEE, 2015.
- [82] Y. Zhang, M. Yang, B. Xu, Z. Yang, G. Gu, P. Ning, X. S. Wang, and B. Zang. Vetting undesirable behaviors in android apps with permission use analysis. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security (CCS '13)*, pages 611–622. ACM, 2013.
- [83] W. Zhou, X. Zhang, and X. Jiang. Appink: watermarking android apps for repackaging deterrence. In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security (AsiaCCS '13)*, pages 1–12. ACM, 2013.
- [84] X. Zhou, S. Demetriou, D. He, M. Naveed, X. Pan, X. Wang, C. A. Gunter, and K. Nahrstedt. Identity, location, disease and more: Inferring your secrets from android public resources. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 1017–1028. ACM, 2013.
- [85] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy (S&P '12)*, pages 95–109. IEEE, 2012.