NORTHWESTERN UNIVERSITY

Towards a Trustworthy Android Ecosystem

A DISSERTATION

SUBMITTED TO THE GRADUATE SCHOOL

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

for the degree

DOCTOR OF PHILOSOPHY

Field of Computer Science

By

Vaibhav Rastogi

EVANSTON, ILLINOIS

August 2015

© Copyright by Vaibhav Rastogi 2015

All Rights Reserved

ABSTRACT

Towards a Trustworthy Android Ecosystem

Vaibhav Rastogi

Mobile devices have become ubiquitous and their sales proportions already exceed the sales of traditional personal computer systems such as desktops and laptops. The number of mobile devices will only increase, and perhaps at an even higher rate in the coming years. Non-traditional architecture, operating environments, and other requirements necessitate rethinking of security and privacy solutions for mobile devices.

Android is the most dominant mobile operating system and so this work makes an effort towards a trustworthy Android ecosystem. We use Android ecosystem as the umbrella term for the interdependent entities such as the Android OS, the applications, application stores developers, devices, the users, and the vendors. While it is challenging to improve security and privacy of the ecosystem owing to conflicting interests of the entities involved, we claim that it is possible to develop malware-detection and privacy-enhancing solutions that can deployed by any party independently of others in the ecosystem. In particular we look at both offline and online schemes that do not require firmware modification, extra privileges, or support from developers. Four separate works are discussed: (a) AppsPlayground, a dynamic analysis sandbox to study the security and privacy properties of Android applications; (b) DroidChameleon, a framework for transforming

Android malware and evaluating anti-malware resistance against transformed malware; (c) AutoCog, a system for inferring the needed application permissions from an application's natural language description; and (d) Uranine, which can be used to detect privacy leakages in Android applications without modifying the Android platform.

Acknowledgments

A lot of credit for my PhD goes to my adviser, mentors, friends, and relatives, without whom this would not have been possible. I would like to expess my gratitude for my advisor, Yan Chen, who led me through this journey of PhD, encouraging me and spending uncountable hours with me on the projects and the issues that came there. Importantly, working with him, I have learnt how to conduct research.

I also have deep appreciation and thanks for all other members of my thesis committee, Profs. V.N. Venkatakrishnan, Robby Findler, and Chris Riesbeck. Prof. Venkatakrishnan has not only been a committee member but an active collaborator and mentor with whom I have had numerous hours of very fruitful discussion. He has guided me not only in my research but also in my career plans. Prof. Findler has always been an inspiring figure from the early days of my PhD when I attended his programming languages seminar. His stimulating teaching style and attention to details, including guidance in formulating my thesis, is something I will always remember. And finally, I would like to thank Prof. Riesbeck for agreeing to be on my committee and giving valuable feedback during the PhD exams.

I would next like to sincerely acknowledge my other collaborators. I learnt a lot about conducting research from Prof. Xuxian Jiang and Prof. William Enck, with whom I published a few of my early works. I also duly acknowledge Prof. Shihong Zou, and Prof. Ryan Riley with whom I have ongoing collaborations at the time of this writing.

The journey of PhD would be quite daunting if we did not have friends around with whom we could share our happiness and distress and feel comforted by their presence. My gratitude goes to my present and former colleagues and friends Zhichun Li, Kai Chen, Yinzhi Cao, Hongyu Gao, Xitao Wen, Zhengyang Qu, Xiang Pan, and Yang Hu, many of whom have also been my collaborators. And then there are other, even more personal friends, some of whom have provided strong emotional support throughout my stay at Northwestern. I am very deeply grateful to my former room-mate and friend forever, Ankit for being such a deep friend.

Finally, I am deeply indebted to my parents who brought me up with lot of love and despite their wishes allowed me to pursue a PhD. I also thank my younger sister for her loving, cheerful, encouraging spirit.

Table of Contents

ABSTRACT 3

Acknowledgments 5

List of Figures 10

List of Tables 12

Chapter 1. Introduction 14

1.1. AppsPlayground 17

1.2. DroidChameleon 18

1.3. AutoCog 19

1.4. Uranine 19

1.5. Organization 20

Chapter 2. AppsPlayground: Automatic Security Analysis of Smartphone Applications 21

2.1. Introduction 21

2.2. Android Background 24

2.3. AppsPlayground Overview 26

2.4. Detection Techniques 31

2.5. Disguise Techniques 33

2.6. Automatic Exploration Techniques 34

2.7. Implementation 40

2.8. Findings 40

2.9. Effectiveness of Automatic Exploration 45

2.10. Related Work 48

2.11. Conclusion and Future Work 50

Chapter 3. DroidChameleon: Evaluating Android Anti-malware against Transformation Attacks 51

3.1. Introduction 51

3.2. Background 54

3.3. Framework Design 57

3.4. Implementation 67

3.5. The Dataset 69

3.6. Results 71

3.7. Defense against Transformation Attacks 77

3.8. Related Work 80

3.9. Conclusion 83

Chapter 4. AutoCog: Measuring the Description-to-permission Fidelity in Android Applications 84

4.1. Introduction 84

4.2. Background and Problem statement 88

4.3. System Design 91

4.4. Implementation 102

4.5. Evaluation 103

4.6. Discussion 115

4.7. Related Work 116

4.8. Conclusion 118

Chapter 5. Uranine: Real-time Privacy Leakage Detection without System Modification for Android 119

5.1. Introduction 119

5.2. Background and Problem Statement 123

5.3. Uranine Design 128

5.4. Implementation 135

5.5. Evaluation 139

5.6. Discussion 147

5.7. Related Work 149

5.8. Conclusion 151

Chapter 6. Conclusion 152

References 154

List of Figures

1.1 Some of the entities involved in the Android ecosystem and the proposed solutions. AppsPlayground may be deployed by application stores or third parties that wish to analyze applications. DroidChameleon evaluates and proposes improvement in security products. AutoCog may again be deployed by application stores or third parties to provide users of the description-to-permission fidelity of applications. Uranine-instrumented applications can be directly run on the devices; the instrumentation service may be provided by any third party. 16

2.1 A simple application with three windows. Window (a) invokes window (c) which invokes window (b). (c) shows only the lower half of the screen emphasizing the menu window. 27

2.2 The GUI hierarchy for the window in Figure 2.1a 27

2.3 Architectural overview of AppsPlayground analysis framework 28

2.4 Overview of the intelligent execution module of Playground 35

2.5 Percentage difference in code coverage between Playground and GUI Ripper 47

3.1 Evaluating anti-malware 71

3.2 An example evasion. Changes required in AndroidManifest of Plankton to evade AVG (original first and modified second; only relevant parts are shown with differences highlighted). No other changes are required. The application will not work though until

the components are also renamed in the bytecode. We confirm that AVG's detection is based on the contents of AndroidManifest alone (see Finding 2). 76

4.1 Overall architecture of AutoCog 91

4.2 Example output of Stanford Parser 94

4.3 Flowchart of description-to-permission relatedness (DPR) model construction 97

4.4 Interpretation of metrics in evaluation 108

4.5 Histogram for distribution of questionable permissions 114

5.1 Deployment by Vendor or Third-party Service 124

5.2 A depiction of challenges C1 and C2 met in Uranine. There are paths between app code and framework code depicted as meandering function call paths and return paths, together with callbacks (the app code that is called by framework code). The left path results from ordinary calls while the right path includes callbacks. Information flow tracking can only be done for app code, requiring approximations for framework code. Callbacks must be handled soundly. Objects on the heap point to each other and their effect on information flow should be properly accounted for during approximations. 126

5.3 Instrumentation flow in Uranine 128

5.4 Associating taint data-structures with objects 133

5.5 Uranine implementation depicting the use of existing code (white boxes) and the features we implemented (gray, discussed in detail in Section 5.3). 135

5.6 Instrumentation organization 137

List of Tables

2.1 Private Information Leaks Detected 41

2.2 Most common leaking domains. The percentages indicate the proportion of apps which leak the corresponding information. 42

2.3 Malware samples used for testing anti-malware tools 44

3.1 Anti-malware products evaluated. 68

3.2 Malware samples used for testing anti-malware tools 68

3.3 Key to Tables 3.4, 3.5 and 3.6. Transformations coded with single letters are trivial transformations. All others are DSA. We did not need NSA transformations to thwart anti-malware tools. 72

3.4 DroidDream transformations and anti-malware failure. Please see Table 3.3 for key. 'x' indicates failure in detection. 73

3.5 Fakeplayer transformations and anti-malware failure. Please see Table 3.3 for key. 'x' indicates failure in detection. EE transformation does not apply for lack of native exploit or payload in Fakeplayer. 73

3.6 Evaluation summary. Please see Table 3.3 for key. '+' indicates the composition of two transformations. 74

3.7 Summary of results from anti-malware tools downloaded in February 2012. Please see Table 3.3 for key. '+' indicates the composition of two transformations. Results that have changed since then are depicted in bold (see Table 3.6 for comparison). 76

4.1 Distribution of noun phrase patterns used 96

4.2 Permissions used in evaluation over 37,845 applications 105

4.3 Statistics and settings for evaluation 107

4.4 Results of evaluation 109

4.5 Example semantic patterns 113

4.6 Correlation between application popularity and the number of questionable permissions and permissions requested. All values are statistically significant with $p < 0.001$ 114

5.1 A comparison of Uranine with dynamic approaches. + is better, $-$ is worse. 121

5.2 Accuracy evaluation of Uranine and comparison with TaintDroid 139

5.3 Leaks detected in automatic tests 143

5.4 CaffeineMark 3 performance. This benchmark may not adequately represent the real-world performance of Uranine 145

5.5 Macrobenchmark performance. The reported times (Original/Instrumented columns) are medians over five independent runs. 146

CHAPTER 1

Introduction

Mobile devices have become increasingly ubiquitous and their sales proportions already exceed the sales of traditional personal computer systems such as desktops and laptops. The number of mobile devices will only increase, and perhaps at an even higher rate in the coming years. Mobile devices are often always-on, always with the user, and carry a lot of user's private data. The security of these devices is thus an important concern.

Security design and issues in mobile devices differ from those in traditional personal computers. Mobile devices hold much more private data for the user, such as their address books, call logs, and so on. Being always with the user, there are added risks such as location tracking. Furthermore, smartphones provide services, like calling and text messaging, exploitation of which could cost the user money. Added to these are security designs for new operating system architectures. There are further issues that need to be dealt with: not hogging the device's battery is one example.

We use 'smartphone ecosystem' as an umbrella term for the interdependent entities such as the smartphone operating system and its developer, the devices, the applications, application stores, application developers, devices, the users, the device vendors, the cellular carriers, and so on. Enforcing security in an ecosystem would require deployment of solutions at multiple points in this ecosystem, usually with support of many different entities involved in the ecosystem. This makes the problem challenging – entities such as the consumers, vendors and the developers may act oblivious to the interest of other entities.

One model to enforce security is that with a centralized authority, for which Apple's iOS devices serve one of the best examples. The central control of Apple allows for greater policing as to what content reaches the consumers. Applications may only be installed from the Apple-controlled AppStore, and there are strict policies enforced in the review of those applications.

An alternative model is that adopted by Android. Android is the most dominant smartphone operating system, primarily led by Google. It is based on the Linux kernel and provides a middleware implementing subsystems such as telephony, window management, management of communication with and between applications, managing application lifecycle, and so on. Due to a permissive open source license and liberal policies, it affords an open model, whereby device vendors and carriers can customize the OS and use it in their devices, developers can create and deploy applications with few restrictions, and consumers can install applications from any source. This openness drives innovation and competition and hence may be seen as a good thing. However, it comes at a price – despite the security features included in Android as a modern operating system, user privacy and malware have become a dominant concern. As mentioned above, it is not easy to enforce security in a decentralized ecosystem where entities may have conflicting interests. This work hinges on enabling adequate security and privacy properties in such an ecosystem without sacrificing the desired openness. Specifically, we make the following thesis statement:

> *It is possible to develop malware-detection and privacy-enhancing solutions that can be deployed by any party in the Android ecosystem independently of other parties.*

We propose and describe security and privacy solutions in the open, decentralized model provided by the Android ecosystem where one may not expect support for a solution from all entities

Figure 1.1. Some of the entities involved in the Android ecosystem and the proposed solutions. AppsPlayground may be deployed by application stores or third parties that wish to analyze applications. DroidChameleon evaluates and proposes improvement in security products. AutoCog may again be deployed by application stores or third parties to provide users of the description-to-permission fidelity of applications. Uranine-instrumented applications can be directly run on the devices; the instrumentation service may be provided by any third party.

of the ecosystem. Yet it should be possible for the interested parties to deploy and take advantage of such solutions.

We now discuss the design space for such solutions. Both online (real-time on-the-device detection and monitoring) and offline solutions are possible. Offline solutions ideally fit these requirements: they can be deployed off-the-device and the results collected from there can then be used to make decisions about security and privacy on the device. It is also be possible to convert some online solutions into offline solutions. Finally, there are online solutions that do not require firmware modifications, special privileges from the operating system, or support from other application developers.

We look at both online and offline malware-detection and privacy-enhancement solutions without requiring any firmware modifications, special privileges from operating system, or support

from developers. Four separate works are discussed (Figure 1.1 identifies where these works are deployed in the Android ecosystem), (a) *AppsPlayground*, a dynamic analysis sandbox deployable at application markets or by third parties to study the security and privacy properties of Android applications; (b) *DroidChameleon*, an evaluation of current anti-malware resistance against obfuscations in malicious applications and proposal of solutions that may be implemented by security vendors; (c) AutoCog, a system for inferring the needed application permissions from an application's natural language description; and (d) *Uranine*, a work in progress, which can be used to detect privacy leakages in Android applications without modifying the Android OS and which can be deployed by application markets or directly on device, possibly backed by off-device cloud services, by the interested users. These works are by no means exhaustive; they however cover the design space discussed above. AutoCog is primarily an offline analysis system whereas AppsPlayground is a general tool to convert many online analysis techniques into offline techniques. Uranine is primarily online, real-time technique with a specific design goal of not modifying the Android OS or needing higher privileges than those available to normal applications. Finally, DroidChameleon also deals real-time on-the-device malware scanning of applications and suggests the use of static analysis, which does not require any firmware modification or higher privileges.

1.1. AppsPlayground

Today's smartphone application markets host an ever increasing number of applications. The sheer number of applications makes their review a daunting task. AppsPlayground is a framework that automates the analysis of smartphone applications. AppsPlayground integrates multiple components comprising different detection and automatic exploration techniques for this purpose. Such a system could easily be deployed at an application market to analyze all applications being submitted. Third parties, such as security vendors, may also deploy AppsPlayground to provide

an independent evaluation of applications published at Android application markets. We evaluated the system using multiple large scale and small scale experiments involving real benign and malicious applications. Our evaluation shows that AppsPlayground is quite effective at automatically detecting privacy leaks and malicious functionality in applications.

1.2. DroidChameleon

Mobile malware threats, especially on Android, have recently become a real concern. It is important to measure the available defense against mobile malware threats and propose effective, next-generation solutions. We evaluate the state-of-the-art commercial mobile anti-malware products for Android and test how resistant they are against various common obfuscation techniques (even with known malware). We developed DroidChameleon, a systematic framework with various transformation techniques, and used it for our study. Our results on ten popular commercial anti-malware applications for Android are worrisome: none of these tools is resistant against common malware transformation techniques. Finally, in the light of our results, we explore possible remedies for improving the current state of malware detection on mobile devices. The general understanding has been that operating system must provide additional support to anti-malware tools to detect malware threats. While we agree with this, we point out that even without operating system support, we can improve state-of-the-art by basing detection on malware semantics rather than syntactic patterns in malware. Such transformation-resilient solutions can be developed independently by an anti-malware vendor without requiring central support, such as from operating system developers or device vendors.

1.3. AutoCog

The booming popularity of mobile devices is partly a result of application markets where users can easily download wide range of third-party applications. However, as discussed, due to the open nature of markets, especially on Android, there have been several privacy and security concerns with these applications. On Google Play, as with most other markets, users have direct access to natural-language descriptions of those applications, which give an intuitive idea of the functionality including the security-related information of those applications. Google Play also provides the permissions requested by applications to access security and privacy-sensitive APIs on the devices. Users may use such a list to evaluate the risks of using these applications. To best assist the end users, the descriptions should reflect the need for permissions, which we term *description-to-permission fidelity*. We present AutoCog, a system to automatically assess description-to-permission fidelity of applications. AutoCog employs state-of-the-art techniques in natural language processing and our own learning-based algorithm to relate description with permissions. In our evaluation, AutoCog outperforms other related work on both performance of detection and ability of generalization over various permissions by a large extent. On an evaluation of eleven permissions, we achieve an average precision of 92.6% and an average recall of 92.0%. Our large-scale measurements over 45,811 applications demonstrate the severity of the problem of low description-to-permission fidelity. AutoCog helps bridge the long-lasting usability gap between security techniques and average users.

1.4. Uranine

The wide range of third party applications on modern mobile platforms enrich the environment and increase usability. There are however privacy concerns centered around these applications – users do not know what private data is leaked by the applications. Previous works to detect

privacy leakages are either not accurate enough or require operating system changes, which may not be possible due to users' lack of skills or locked devices. We present Uranine[1], a system that instruments Android applications to detect privacy leakages in real-time. Uranine does not require any platform modification nor does it need the application source code. We designed several mechanisms to overcome the challenges of tracking information flow across framework code, handling callback functions, and expressing all information-flow tracking at the bytecode level. Uranine further includes static analysis to instrument only paths along which privacy leakage may happen. Our evaluation of Uranine shows that it is accurate at detecting privacy leaks and has acceptable performance overhead.

1.5. Organization

The rest of this prospectus is organized as follows. Chapter 2 describes AppsPlayground. Chapter 3 presents the evaluation results of our DroidChameleon and the proposed solutions for improving the present situation. Chapter 4 presents AutoCog while Uranine is presented in Chapter 5. Finally, a conclusion is presented in Chapter 6.

[1]Uranine is a dye, which finds applications as a flow tracer in medicine and environmental studies.

CHAPTER 2

AppsPlayground: Automatic Security Analysis of Smartphone Applications

2.1. Introduction

Mobile devices such as smartphones have gained great popularity in response to vast repositories of applications. Most of these applications are created by unknown developers who may not operate in the users' best interests, leading to malware [5, 92] and frequent exposure of privacy sensitive information such as phone identifiers and location [46, 48, 49].

Recently, researchers have proposed both static and dynamic security analysis techniques for smartphone applications. While static analysis approaches such as those used by PiOS [46] and Enck et al. [49] scale to large numbers of applications, they do not capture runtime environment context such as configuration variables and user input. More importantly, code may be obfuscated to thwart static analysis, either intentionally or unintentionally (such as stripping symbol information of native binaries to reduce size).

On the other hand, TaintDroid [48] uses dynamic analysis to capture runtime environment context. However, the researchers had to manually navigate the user interfaces of each analyzed application to sufficiently exercise dangerous functionality. More recently, DroidScope [142] used dynamic analysis for malware forensics. Large-scale dynamic analysis however requires more than what has been proposed earlier – a fast analysis system and strategies to provide automatic code coverage.

In this chapter, we propose *AppsPlayground*, referred to as simply *Playground* for brevity, a framework for automated dynamic security analysis of Android applications. Playground is meant

to analyze applications for both malware, i.e., apps that have a malicous intent, and grayware, i.e., apps that are not malicious but may still be annoying, for example, by leaking private information for a legitimate purpose but without user's awareness. From this point on, for the sake of conciseness, we will not particularly distinguish between malware and grayware and refer to them both as malware. An automatic dynamic analysis framework needs to possess not only detection techniques for identifying malicious or annoying functionality but also automatic exploration techniques to explore the application code as much as possible. Furthermore, the dynamic analysis environment needs to appear as real (in this case, a real smartphone) to the app as possible, lest a malicious app can easily detect the special environment and not show any malicious behavior.

In Playground, solutions to all the above requirements are integrated together in a modular manner. We use multiple detection techniques, ranging from taint tracing to kernel-level system call monitoring. For taint-tracing, we are able to seamlessly integrate and reuse TaintDroid [48], an already available taint-tracing engine with very good performance for Android into the rest of our system. In order to deal with root attacks in Android, we describe vulnerability conditions in Android as succint signatures in terms of system calls and kernel-level data structures. These signatures may easily be incorporated into a dynamic analysis.

For automatic exploration, we find that the nature of Android imposes non-conventional requirements on the exploration techniques that need to be used. Application code can be triggered by several kinds of system events and so such events need to be simulated. Moreover, most of the apps in Android provide GUI, which requires sophisticated GUI exploration schemes. Trivial approaches for GUI exploration such as fuzz testing have their advantages in their simplicity and, if designed properly, have the ability to eventually exhaustively explore a finite state space. They however take more time and are sometimes insufficient because application user interfaces have complex requirements such as login credentials for Internet services. Therefore, we also need to

intelligently drive the user interface to exercise code implementing interesting and dangerous functionality. Our heuristic-based intelligent execution technique is able to avoid redundant exploration and is able to use contextual information to fill editable text boxes meaningfully.

To demonstrate the practical advantage of Playground, we evaluated 3,968 from the official Android Market (now Google Play). We identified exposures of privacy sensitive information in 946 applications, flagged by the taint-tracing engine. Of these, 844 applications leaked phone identifiers (such as phone number and IMEI), and 212 applications leaked geographic location. We note that detecting privacy violations still requires manual confirmation, as TaintDroid only identifies that information has left the device over the network interface, and not privacy violations. For further validation, we also tested the applications used in the TaintDroid study. Playground's findings almost completely coincided with the findings manually made by the TaintDroid authors on the much smaller set of thirty applications they evaluated. Furthermore, we also evaluated Playground on known malware samples, falling under diverse categories of root attacks and SMS trojans, and were able to detect the malicious nature of all of them.

Finally, to evaluate the performance of automatic GUI exploration, we compare our system with GUIRipper [96], a system that automatically generates test cases based on windowing elements in traditional desktop GUIs. To the best of our knowledge, this is the only system, apart from fuzz-testing, available in the literature for GUI exploration. It lacks advanced techniques such as filling in contextual data in text boxes and repeatedly exercising GUI widgets to achieve better code coverage, both of which we have found are often critical requirements when testing Android applications. Our comparison with an Android port of this system shows our technique to achieve a mean 30% improvement in terms of code coverage.

To summarize, this work makes the following contributions.

- We propose AppsPlayground (or simply, Playground), a modular framework for scalable dynamic analysis of Android application.
- We identify the key requirements for automatically exploring Android applications. We use automatic system event triggering and propose and develop a new intelligent execution technique that can use contextual information to provide meaningful textual input.
- We describe vulnerability conditions for known vulnerabilities in Android as succint signatures that may be used in dynamic analysis. These vulnerability conditions are necessary for a system compromise.
- We implement the AppsPlayground framework for Android and evaluate 3,968 applications from the official Android app Market. Our analysis identified exposures of privacy sensitive information in 946 applications. Moreover, we were able to confirm the malicious nature of already known malware samples using this framework.

The remainder of this chapter proceeds as follows. Section 2.2 provides relevant background in Android and Section 2.3 gives an overview of Playground. Sections 2.4, 2.5 and 2.6 provide detailed discussion of the techniques incorporated into Playground. Section 2.7 discusses the implementation of Playground. Section 2.8 describes our measurements with Playground. Section 2.9 discusses the effectiveness of the automatic exploration techniques employed. Section 2.10 presents related work and Section 2.11 concludes.

2.2. Android Background

Android is a widely popular and open source operating system designed for smartphones and other mobile devices. While Android is based on Linux, it defines an entirely new middleware and GUI environment in which applications execute. Applications are mostly written in Java, which is

compiled to Dalvik bytecode, which runs in a virtual machine similar to the Java virtual machine. Apart from Java, Android also allows parts of apps to be coded in native code.

Every Android application runs as an unprivileged user with Linux UIDs effectively being used to provide application sandboxes. Android applications are composed of *components*. There are four component types: *activity*, *service*, *broadcast receiver*, and *content provider*. The user interface is defined by one or more activity components. Services are meant to run in background while content providers manage access to data. Broadcast Receivers are registered with system services and can receive system events, such as reboot completed, or an SMS received, and so on. Once a broadcast receiver is registered to receive a system event, the code specified in the broadcast receiver is run whenever the system event is triggered.[1] Most system events are guarded by permissions, which the app must declare and get approved for at installation time.

For automatic exploration, it is necessary to understand the GUI features in Android. Each activity corresponds to a screen displayed to the user. This screen is functionally equivalent to a traditional GUI window, the only difference being that only one screen is shown at a time (with minor exceptions), whereas traditional GUIs can typically display multiple windows.

An application's GUI consists of several activities that invoke one another and possibly return results. At any point in time, only one activity has input focus and processing. This activity is referred to as the *active* activity. When one activity invokes another, the former is paused and the new activity is pushed to the top of the activity stack and made active. Once an activity has completed its work, it terminates, optionally returning a value, and the next activity on the stack is made active. Note that activities are not limited to invoking activities within the same application. A sequence of related activities on the stack is called a *task*.

[1]This may sometimes not hold due to, for example, abort of a broadcast.

The activity GUI layout is commonly defined in XML but may also be defined programmatically. As in traditional GUIs, an Android window consists of widgets, which are are referred to as *views* in Android terminology. The Android library supplies several useful views which may either be standalone (e.g., buttons) or act as containers for other views. In addition to the window layout, an activity can define a menu that appears when the user presses the physical "Menu" button on the phone.

Example. Figure 2.1 shows a simple example application. The application consists of two activities, "Hello World" and "About" (Figures 2.1a and 2.1b, respectively). The "Hello World" activity has three buttons which bring up the "Hello World!!" message in three different languages. The "About" activity is non interactive. There is a menu attached to the "Hello World" activity, which we model as a separate window. After opening this menu, one may click on the only option (named "About") to go to the "About" activity. Figure 2.2 depicts the GUI hierarchy of the window in Figure 2.1a.

2.3. AppsPlayground Overview

This section gives a broad view of Playground. We begin with describing the overall architecture of Playground followed by the different components involved in brief.

2.3.1. Overall Architecture

We seek to design a general framework for automatic dynamic analysis for smartphone applications. Playground is built as a virtual machine environment. Specifically, it repurposes the Android emulator, available with the Android SDK, for the dynamic analysis environement. Built on Qemu [11], the emulator emulates an ARM machine and provides support for a few features available on a real phone, such as telephony.

(a) (b) (c)

Figure 2.1. A simple application with three windows. Window (a) invokes window (c) which invokes window (b). (c) shows only the lower half of the screen emphasizing the menu window.

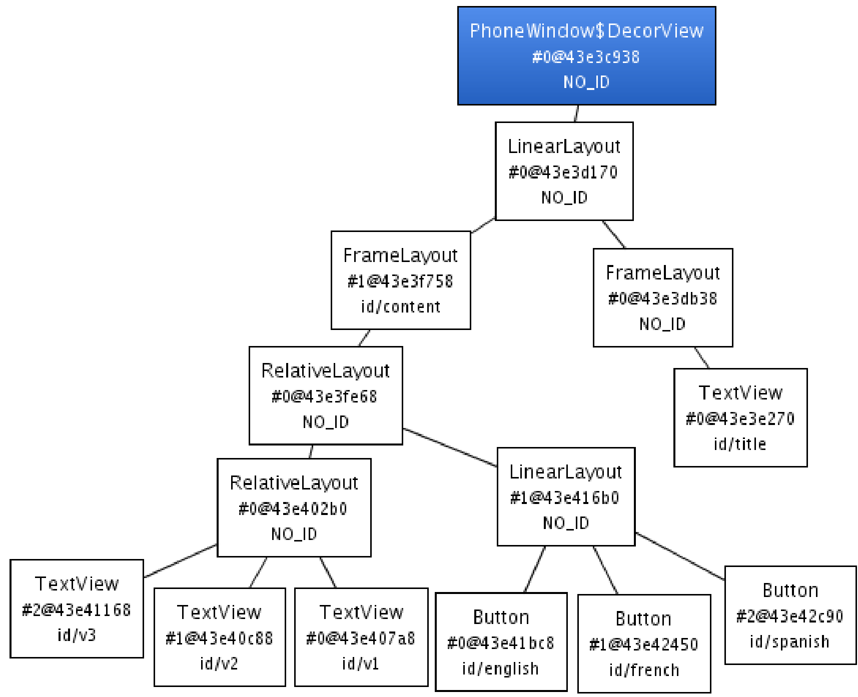


Figure 2.2. The GUI hierarchy for the window in Figure 2.1a

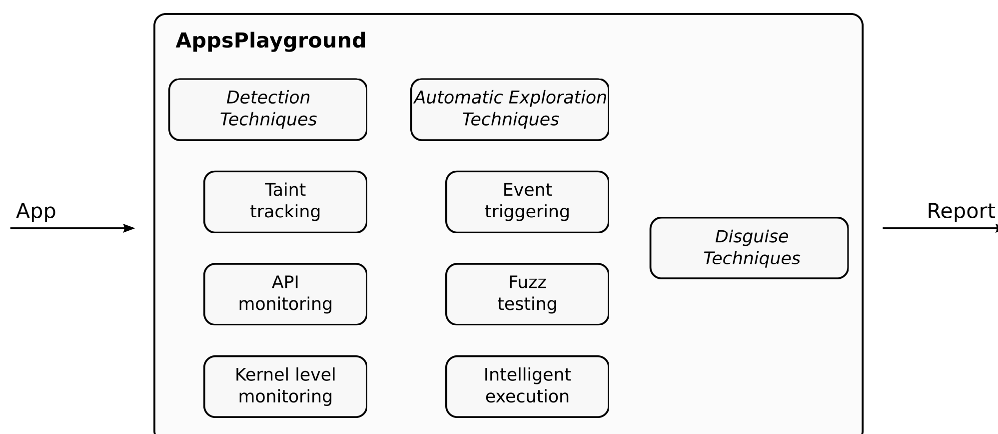


Figure 2.3. Architectural overview of AppsPlayground analysis framework

A virtualized environment is essential to providing scalability required for malware analysis. For example, every analysis can use a fresh snapshot of the environment without affecting the analyses of other samples; this is not feasible when using real devices. However, different from a few past approaches [142], we do not employ virtual machine introspection, a technique in which the virtual machine (VM) guest is run unmodified and any analysis tools run outside the VM, analyzing its physical memory to get information from inside the virtual machine. This approach while complicated, allows the analysis tools to be strictly more privileged than the analyzed environment.

In the case of Android however, apps typically run as unprivileged users and hence introspection is not actually required. Even for known attacks that try to get root privileges, signatures may be developed for identifying the attack and safely recording it before the privilege escalation actually completes. For apps requiring root (through su), these arguments do not apply; however, the number of such apps is low and the number of rooted devices is also significantly smaller. Furthermore, the complexity of introspection also hinders in the retrieval of GUI information or sending events from outside the emulator.

Figure 2.3 shows the architecture of Playground. Playground has several components comprising multiple detection techniques, multiple automatic exploration techniques, and techniques to

make analysis environment appear like a real phone. All these components work independently of each other and integrate together in a plugin-able manner. We next briefly discuss the components listed in the figure.

2.3.2. Playground Components

Detection techniques are the components that actually provide the detection of a possibly malicious functionality while a sample is being tested. The detection techniques that we include are taint tracing for information leakage detection, based on TaintDroid; sensitive API monitoring, such as monitoring for the SMS API; and kernel-level monitoring for detection of root exploits. Disguise techniques are those that make the environment appear like a real device; these include the use of realistic phone identifiers, keeping realistic data in phone databases, and so on.

Automatic exploration techniques help in automatically increasing code coverage of the application code. Without automatic code coverage, it is likely that much of the code in an application will not be executed. Playground simulates events, such as location change and sms received, to trigger code in event receivers (primarily broadcast receivers). To explore the app GUI, we use fuzz testing and intelligent black-box execution. Since fuzz testing simply sends in a stream of random inputs, it may be described as a random walk on the state space. Given the ability to restart from the start state any number of times, it can eventually explore any finite connected state space. Applications that do not need any meaningful text to be filled in have a small state space consisting of screen taps and drags. Fuzz testing can deal with such applications quite well without any knowledge of their interaction model. On the other hand if some meaningful texts such as login credentials are required, fuzz testing cannot enter in the right input, and fails. For such cases, we need intelligent execution, which heuristically determines what data has to be entered in.

Furthermore, since fuzz testing is random, it may sometimes fail to explore some states. Intelligent exploration however deterministically explores states that it can model.

Intelligently driving the user interface of smartphone applications presents several challenges:

- *Modeling the GUI.* In order to intelligently exercise the user interfaces of applications, a representation of the program flow must be abstracted from the GUI. The closeness of this approximation to the actual program flow determines the completeness of the automation algorithms.
- *Efficient exploration strategy.* Even simple applications can have a very large (if not infinite) number of unique program states based on user input (e.g., a counter). Practical testing of applications requires an efficient exploration strategy with the ability to effectively discover distinct and useful states and remove redundant states.
- *Context determination.* Applications often have text fields that require special values. Leaving them empty or filling in garbage can limit application exploration. A few real world examples follow.
  - *Login credentials.* Unless a correct username and password is supplied in the correct fields, the exploration of the application will be seriously limited.
  - *Cities and zip codes.* Application functionality depending on zip codes and cities entered in input fields will likely fail in the presence of random input.
  - *Duplicate input fields.* Applications occasionally require the user to enter the same information in two text fields for consistency checks, e.g., passwords, PINs, and Email addresses.
  - *Input format.* Fields such as Email addresses and phone numbers are occasionally required to be entered in a specific format before the application will accept the input.

– *Dates.* A future date may not work when a past date is expected. An application which asks for date of birth may not move forward if a date is in the past but is one that does not indicate the user is now over 13.

In all these cases, Playground must infer from the context present around text fields what should be filled in. We note in most cases, these inputs are validated by remote servers and so even symbolic execution cannot help determine correct values for them.

2.4. Detection Techniques

In this section we discuss the various detection techniques that are included in Playground. Other techniques may be included as needed.

Taint tracing. Playground uses taint tracing to track privacy-sensitive information leakage. We have integrated a slightly modified version of TaintDroid [48], an open-source, high-performance taint-tracing system for Android. We note that TaintDroid works only for Dalvik bytecode only. Native code taint-tracing would likely require dynamic binary instrumentation or VM instrospection. We currently do not use such methods for native code taint-tracing; these methods result in a typical slowdown of 10x to 30x for the code and hence are not very attractive from the performance perspective.

Sensitive API monitoring. Playground monitors a few system APIs for detecting possibly malicious functionality. The SMS API is one of the most exploited API in Android. Malicious apps use it to send text messages to premium rate numbers without user's awareness. Playground can record the destination and content of the SMS messages sent by an app. Similarly, Playground monitors the Java reflection API to record method calls and field accesses through reflection as some of these may be indicative of obfuscated codes typical in malware. Playground also monitors dynamic bytecode loading and can inform the analyst of which bytecodes (contained in a .dex file)

were loaded. We note that monitoring reflection and bytecode loading APIs is done for application code only. Framework code is trusted and so need not be monitored. The differentiation is done on the basis of class loaders; in Android the class loaders for application code are always different from the class loader that loads the framework code.

Kernel-level monitoring. We also provide kernel-level tracking to identify known root-exploits. Our method of identification of root exploits is based on vulnerability conditions and is thus immune to code polymorphism. We observe that known root exploits such as rageagainstthecage, exploid, and gingerbreak, all have signatures that can easily be used in dynamic analysis without raising too many false alarms:

- Rageagainstthecage/Zimperlich. These attacks fork RLIMIT_NPROC (the maximum allowable) number of processes for a UID (the UID associated with the malicious app) and then make zygote (a system daemon) spawn another process for that user. The zygote daemon typically uses setuid system call to change the UID to the app's uid. However, since this UID already has as many processes as are allowed, setuid fails, and the app gets a process with root privileges. We observe that this attack can be detected simply by monitoring if the number of processes for a user comes close to the maximum allowed.
- Exploid (CVE-2009-1185). This exploit is based on a vulnerability in the init, in which init does not check the origin of NETLINK messages. Untrusted code may thus be registered and get called later. For this vulnerability to happen, a neccessary condition is that the app code must send a NETLINK message later. We can use this as our signature.
- Gingerbreak (CVE-2011-1823). This exploits a vulnerability in the vold daemon in Android, again requiring untrusted code to send NETLINK messages to vold. Hence our signature here is similar to that for exploid.

We note that the above three are representative examples. In general we can encode conditions for any vulnerability in code; the checks will be inserted in the crtical path that leads to the given vulnerability. We note that the OS used for analysis need not actually be vulnerable for the vulnerability conditions to get triggered. Hence, attacks for vulnerabilities in multiple versions of Android may be detected on the same version. Moreover, attacks that would normally not succeed in the emulator may also be detected.

2.5. Disguise Techniques

Playground adopts a number of measures to make the analysis environment appear realistic. It provides real-looking phone identifiers to the app. These identifiers include the phone number, IMEI, IMSI, Android ID and so on. We also modify the build.prop (a file that contains several properties about the system) properties to match a real device. In a similar vein, we can also modify identifiers that relate to Qemu and other virtualization-related features.

Furthermore, we provide realistic data on the device, such as contacts, SMS, pictures, files on SDCard, and so on. We also provide additional libraries such as the Google Maps library, which is available on real devices. In addition Google apps (a set of Google proprietary apps available on a majority of Android devices) may also be provided though we do not provide them at this moment. Data from sensors such as GPS is also made to appear realistic. Currently, we do not support all sensors. Support for microphones is partial while we do not have any support for accelerometers.

We note that evasion of virtualized environments has long been an issue. Even if the above problems are fixed, there will always be evasion techniques based on timing (virtual devices run slower) and Qemu fingerprinting, for example [117]. These problems are general to all dynamic analysis systems.

2.6. Automatic Exploration Techniques

We discuss here the techniques used for automatic exploration in Playground. The next two subsections describe event triggering and intelligent execution. Fuzz testing being almost a trivial technique is skipped from discussion here. Currently, Playground does not use any symbolic execution, which appears to be a good option for state space exploration of an app. We note that there are presently no effective symbolic execution solutions for interactive applications such as those involving GUI. Even projects developed around symbolic execution use random walks or fuzz testing to explore the GUI parts of the applications [125]. Symbolic execution can however be used to make event triggering better. For example, SMS messages received from only certain numbers may trigger some code in the application; symbolic execution could be used to construct the right kinds of messages here. We plan to include symbolic execution into Playground as a future work.

2.6.1. Event Triggering

Several API elements in Android are event based. Applications may register some code to be triggered whenever an event happens. There are specific events raised by the system when, for example, an SMS is received, the device location changes, the system completes a reboot, a call is received or is hung up, and so on. Sensitive events are guarded by permissions, which an app must declare statically and get approved for at the time of installation. Many malicious applications have been found to register for specific events [148].

Based on the permissions declared by the application, we raise specific events in the system. For example, if an application contains the BOOT_COMPLETED permission, Playground artificially raises the reboot completed event (note that we use VM snapshots only; booting the VM will

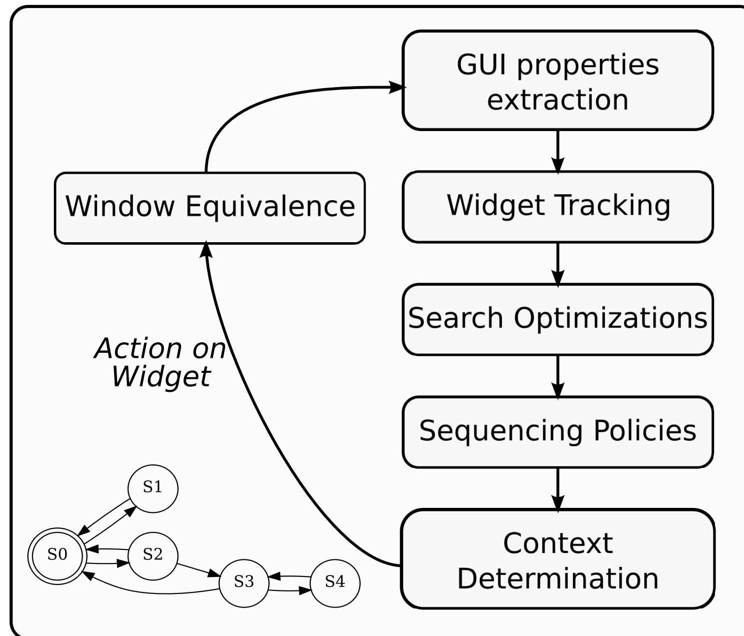


Figure 2.4. Overview of the intelligent execution module of Playground

be much more time consuming). This triggers the app's code that was registered with this event. However, artificially raising important events may cause system inconsistencies as well. This happened with the reboot completed event. We correct some of the framework code so that it would react to this event only once. Other events are handled similarly.

2.6.2. Intelligent Execution

Playground intelligently drives the user interface of a smartphone application by dynamically defining and exploring a model created from window and widget features. We extract features from displayed user interfaces to iteratively define a model that approximates the application's logic. For example, when an application launches, it displays a window with one or more buttons. When a button is selected, a new window appears. The transitions between windows are captured by this model. Note that this approach is based on the intuition that smartphone applications are

highly interactive and that the resulting model provides a good approximation of the application's logic states.

Figure 2.4 presents an overview of the intelligent execution module. For every iteration, Playground checks if focus has changed to a different window. To avoid redundant exploration, a window equivalence module uses heuristics to determine if the newly displayed window is similar to previously viewed windows. If so, the window is merged with an existing state. Playground then extracts features relevant to driving the GUI. These include widgets containing texts, editable text fields, buttons, scroll containers and so on. It then creates associations between the current features and those retrieved earlier using widget tracking (why this is needed is discussed below). A few search optimizations are applied next to prune the search space. Next, Playground uses sequencing policies to determine the next GUI action (such as select a button, scroll down, fill text fields). Text fields are filled using heuristics defined by the context determination module. The current iteration is completed with the performance of an action. The rest of this section describes the various modules shown in Figure 2.4 in greater detail.

Widget Tracking. When navigating windows, widgets may disappear and later reappear. Failure to identify a widget when it reappears may result in concluding identical states or events to be different and hence redundant exploration. For example, consider a window with buttons A and B. Upon pressing button A, the window closes. To complete the exploration, the window is re-opened. The problem would be trivial if the each widget has a unique identifier. This is unfortunately not true for Android.

Playground tracks widgets similar to the way a human user might. We have identified the following widget properties for widget tracking. (1) *Text associated with a widget.* Widgets often have some text associated with them which is made visible to the user, e.g., a text label on a button. In many conditions, this text is sufficient to uniquely identify the widget. However, not all widgets

have associated text. Additionally, multiple widgets may have the same text. (2) *Image associated with a widget.* GUI layouts often use widgets containing an image. In such cases, the image can uniquely identify the widget.[2] (3) *Position within the window.* Combined with the previous previous, the location of the widget on the screen is a useful indicator. Finally, (4) *Position in the GUI hierarchy.* Widgets often have fixed chains of ascendents. A button, for example, will always have the same chain of ascendents in a window. The user perceives this in terms of the relative positioning of widgets.

Sequencing Policies. Each window can contain many widgets that allow input events. In addition to buttons, a window can contain editable text boxes, check boxes, spinners, etc. The result of selecting a button can be directly influenced by interaction with other widgets. Check-boxes can enable/disable other widgets. Finally, scrollable container widgets hide other widgets from the user. Exercising every possible sequence of widget interaction is infeasible. So, we have to arrange the order of event execution in the most meaningful way.

The sequence of interaction with widgets in a window requires consideration. Based on observation, we classify GUI input events into two groups: (a) those that input parameters or variables into the app, such as inputting text into an editable text box or a spinner, and (b) those that provide actions, such as buttons. First, widgets that accept input variables should be acted upon before action widgets. Second, widgets that are contained within a scrollable container are acted upon before scrolling the container. Third, contents of the scrollable container and the container itself are exercised before acting upon widgets outside the container, except when this is in conflict with the first rule. This design choice follows the intuition that the widgets outside the scrollable widget (if present) are often the control buttons such as “OK”, “Submit”, and “Cancel”.

[2]We modified Android framework for exporting image identifiers which could be hashes of images, their resource names, and so on.

Note that the choice of these policies has important ramifications. If the behavior of a widget depends on another widget, Playground may not be able to trigger the entire set of behaviors. While we discuss this problem within a single window, it is easy to see such problems would also arise across windows.

Search Optimizations. For the sake of practicality, we heuristically prune redundant navigation paths where possible. For items organized as a list or a grid, we explore the items up to a threshold. In addition to reducing exploration time, a threshold is sometimes necessary to achieve program termination. For example, an Android list may dynamically expand and thereby go infinitely deep. We also put a threshold on the number of times the same widget may be interacted with (interacting with the same widget more than once may be required to completely explore the states that this widget leads to).

Window Equivalence. When exploring an application, a window is often invoked several times with different parameters. For example, consider an address book application. One window displays a list of contacts. When a contact is selected, an "edit contact" window is opened. On selecting different contacts, the resulting window will be similar, but not identical. Similar windows often correspond to the same application functionality and underlying code. Playground reduces the search space by annotating such equivalent windows.

Playground uses window equivalence heuristics to determine if the current window state is sufficiently similar to a previously visited window state. For our Android implementation, we leverage the correspondence between activity components and window design. That is, our heuristic classifies all windows belonging to the same activity component as equivalent. GUI Ripper [96] also used window titles to determine window equivalence.

Context Determination. As previously discussed, applications often have text fields that must be filled with appropriate values to lead them to the right states. Playground searches for keywords

in the hints and the widget IDs[3] associated with editable text boxes and in the visible text labels next to them. For example, the string "Email" may appear immediately to the left of a text box, indicating that it should be filled in with an Email address.

Determining the keyword rules requires empirical investigation. We analyzed the string resources of over 500 Android applications to determine which strings application developers use for particular fields. To do this, we first extracted all of the strings an application's string resource file. We then converted the strings into a canonical form (lowercase, de-hyphenated). Next, we sorted the strings of all applications by frequency. The result was used to manually classify the strings into various semantic buckets, e.g. email, name, and phone. Finally we coded keyword based rules for each semantic bucket. Our final specification included rules for email, address, date, phone number, password, username, cancel, and ok, among several others. The approach of automatically filling in text fields has also been used for web form completion [71, 118]. These techniques are more sophisticated and include self-learning. We plan to integrate these techniques into Playground.

Our strategy for addressing account sign-up and sign-in follows from the keyword rules approach for context determination. Sometimes, an application requiring sign-in will also include a window to sign-up for the service. The sign-up window will contain text input fields for Email, username, and password. By identifying these fields, Playground can automatically sign up for an account if a sign up option is available from within the app. Currently, Playground always uses the same Email address, username, and password; subsequent tests of an application will automatically sign in by filling in the same credentials. In future, Playground may also be able to identify if it could not successfully log in. A human tester can then create an account which Playground can use to automatically test at least future versions of the application.

[3] Developers often tend to give descriptive IDs to widgets which often convey the purpose of those widgets

2.7. Implementation

We have implemented the Playground analysis framework. The implementation is done over the standard Android emulator that comes with the Android SDK. We modify the Android source code to integrate TaintDroid and to insert hooks for API level monitoring. Kernel modifications are made to provide kernel-level monitoring. Furthermore, disguise measures are implemented by changing the appropriate identifiers and data, either directly in the Android source code or by adding files on the disk images and changing the content of the standard databases (such as contacts). Minor changes were required to the Android source for doing event triggering and fuzz testing. Intelligent execution interfaces with the window manager in Android to retrieve window and widget properties from the system. We use the ViewServer/HierarchyViewer for the interface. Changes are made to the code of many standard widgets so that required widget properties may be retrieved. We further modified related code to make retrieval of properties faster than in the original code.

Apart from the guest (Android) side, Playground also has a host side, written in over 3,000 lines of Java code. The host side implements the algorithms for intelligent execution, and also handles the dispatch of apps to multiple emulators for parallel testing and the logging of information received from the detection techniques running inside the emulator.

2.8. Findings

To show the effectiveness of Playground, we conduct some small-scale and a large-scale experiment. Our first experiment tries to automatically derive the results obtained in the TaintDroid paper. The second experiment is conducted on a set of 3,968 apps downloaded from the Android Market in November 2010. Finally, we also test Playground on real, known malware to evaluate the effectiveness of Playground at detecting malware.

Table 2.1. Private Information Leaks Detected

Number of applications	3968
Information type	Number of applications leaking
GPS	212
Android ID (AID)	581
IMEI	329
IMSI	91
Phone number	63
ICC-ID	3
WiFi MAC address	4
All types	946
At least one ID	844
At least one non-AID ID	442
GPS with at least one ID	120

For taint tracing in our experiments, we tracked device identifiers and location information leaks. By device identifiers we mean any strings that may be used to identify a particular device. Android ID is an identifier on Android available to any app without requesting any special permission. IMEI is an identifier available on all GSM phones. IMSI is associated with the SIM card and identifies a user on the cellular network. The ICC-ID is also specific to a SIM card. Access to IMEI, IMSI, ICC-ID, and WiFi Mac address requires special permissions.

2.8.1. Small-Scale Validation

To validate the effectiveness of Playground in helping discover privacy leaks, we used Playground to drive the same set of applications as that studied in the original TaintDroid paper. The TaintDroid researchers had to manually explore the applications but we attempt to achieve the same detection automatically here. Out of thirty total applications, we had to exclude nine because they were now obsolete and non functional or would not run properly on the Android emulator. Of the rest we were able to reproduce the exact findings from the manual tests conducted by the TaintDroid

Table 2.2. Most common leaking domains. The percentages indicate the proportion of apps which leak the corresponding information.

	# uniq apps	# uniq creators	Android_id	IMEI	IMSI	Phone #	Location
data.flurry.com	265	180	98.1%	2.2%	0	0	14.0%
mobclix.com	152	71	95.4%	68.4%	0	0	12.5%
Google related domains	63	58	0	0	0	0	96.8%
localwireless.com	58	1	0	0	100%	0	24.1%
admob.com	51	27	0	0	0	0	90.1%
ad.qwapi.com	45	26	97.8%	2.2%	0	0	13.3%
playgamesite.com	29	2	0	100%	0	0	0
ade.wooboo.com.cn	21	8	100%	0	0	100%	4.7%

authors except in two cases (Wisdom Quotes Lite, Traffic Jam) where location leaks were not detected. In one other case (Babble) however, we detected an additional location leak which was not found in the original TaintDroid experiments. Such discrepancies are possible due to non deterministic behavior of applications which has been witnessed by others also [69]. Moreover, we also detected several leaks of Android ID which was not being tracked in the TaintDroid paper. This experiment thus conclusively establishes the effectiveness of Playground at automatically detecting privacy leaks.

2.8.2. Large Scale Measurements

We used Playground to drive 3,968 applications. Our findings are summarized in Table 2.1. We detected 946 applications to be leaking information to Internet, which is 23.8% of total number of applications we evaluate. This is because many free applications likely include third party ads and/or analytics libraries which track unique users based on these identifiers. Among the identifiers, Android ID is the one with least risk, as it can be changed at any time. Other identifiers are permanently associated with either the device or the SIM card. We find that in 52.3% of applications leaking an identifier, there is at least one non Android ID identifier. In 56.7% of instances of location leaks, both an ID and the location information is leaked out. In these cases,

the applications can uniquely track the location history of the users. We also found 63 phone number leaks. Since phone numbers are often found on social networking profiles, the privacy implications of tracking are more significant than those of other identifiers.

Analysis of Results: We would like to know the final destinations of information leaks; if the leaks are to advertisement/analytics networks or to developer's own servers. Usually, the applications from a single creator[4] may share the same set of servers. If applications from multiple creators leak the information to a single destination domain, it is most likely the domain belongs an advertisement/analytics network, or a domain related to third-party libraries used by the applications. We found a total of 392 unique domains. Of these 29 domains relate to at least two creators. These are more likely to be advertisement/analytics networks. The rest of the domains come from single creators and hence are very likely to be domains used by the developers.

In Table 2.2, we show the domains that are related to a large number of unique applications. We also show what information has been leaked to this domain. For example, we find in 98.1% of leaks to data.flurry.com, the Android ID has been leaked. We find most of these are advertisement/analytics networks. localwireless.com and playgamesite.com are however developer sites. We note that AdMob is known to track users on the basis of hashed device identifiers. TaintDroid does not propagate taint through cryptographic hash functions and hence it appears, that none of the identifiers were sent to AdMob.

2.8.3. Analyses on Known Malware

We also analyzed known malware to confirm that Playground is able to detect malware in the wild. We considered three malware samples, FakePlayer, DroidDream, and DroidKungfu. The first one is an SMS trojan that sends SMS messages to premium numbers. The other two are root exploits.

[4]We obtained the creator information from the Android Market

Table 2.3. Malware samples used for testing anti-malware tools

Family	Package name	SHA-1 code	Date found	Remarks
Fakeplayer	org.me.androidapplication1	1e993b0632d5bc6f074 0ee31e41dd316435d997	08/2010	SMS trojan
DroidDream	com.droiddream.bowlingtime	72adcf43e5f945ca9f7 064b81dc0062007f0fbf	03/2011	Root exploit
DroidKungFu	com.sansec	4bf050f089a0d44d686 ff74b75cb7f1706fdcaa	05/2011	Root exploit

Detailed information about the samples may be found in Table 2.3. Following is our experience of analyzing these malware samples with Playground.

FakePlayer. This malware sample installs as a movie player. On starting the application, the an activity came up momentarily and then closed. On checking the logging done by Playground, we found that this app had sent three text messages to short numbers 3353, 3354, and 3353 in sequence. Each message contained text "798657". The SMS destinations being short would make it highly suspicious that this sample is malware.

DroidDream. On starting the application inside Playground, we did not experience anything suspicious; rather the app crashed. On disassembling the app's code and examining it, it turned out that the app would get stuck on the "phoning home" behavior. Apparently, it tries to connect to a remote server sending private information about the phone, including IMEI and IMSI numbers, but failed when we tested because the remote server did not respond. We removed this "phoning home" behavior (which is a single method call with the name of `postUrl()`), and tested the modified app again. It turns out that this time app did execute the rageagainstthecage exploit. We could see several running processes with this app's UID and finally could also see a root process; the privilege escalation had completed. Next, we checked the logs collected by Playground. The logs showed a huge number of forks and exceeding of a threshold number of processes. The logs thus give sufficient evidence of the rageagainstthecage attack having being attempted.

DroidKungFu. On launching this app inside Playground, the only thing we observed was the "phoning home" behavior, which is quite well documented. The app sent the IMEI, Android version, and phone model out of the phone. While IMEI was explicitly marked to be taint-traced; the Android version and phone model appeared as plain text in the logs as being sent out of the phone. We however did not observe any attempt to gain root privileges. On looking deeper into the code, we found that the root exploits were not executed due to some condition checks, which looked for the existence of `/system/xbin/su` and some version checks. Changing either the analysis environment or the app code would allow us to see the attacks being executed. This is a general problem in dynamic analysis that sometimes the environment conditions may not match. Symbolic execution may be of help here.

2.9. Effectiveness of Automatic Exploration

In this section we evaluate and discuss the effectiveness of automatic exploration. For this, we augmented the Dalvik VM to report code coverage in terms of the number of instructions executed. Next we compare our system with GUI Ripper and then provide a discussion where we include our experience on automatic exploration.

2.9.1. Comparison with GUI Ripper

We compare our system with GUI Ripper [96]. We ported it to Android based on the information available in Memon et al. [96]. Playground is essentially a superset of GUI Ripper. This meant that we simply remove some of the functionality of Playground (such as context determination and repeatedly exercising widgets) to get a GUI Ripper configuration.

For Playground, we observed a code coverage mean of 33%. We observed 27% mean code coverage GUI Ripper. The low coverage is expected because both the systems treat the application

as a black-box. In fact, low coverage is one of the most limiting factors in dynamic analysis. It is also true that many applications may not give close-to-100% coverage. There may be several reasons for this. Applications may have dead code or code which executes only under special circumstances such as special device configurations and so on.

To get a comparison between Playground and GUI Ripper, we (a) disregard instructions executed by simply starting the application (since these instructions are trivially executed without the need of any navigation), and (b) calculate the *percent difference* between Playground and GUI Ripper. Since, we are interested in the cases when Playground performs better (or worse) than the other approaches, we do not use the absolute value of the difference, i.e., we use $C(x, y) = \frac{(x-y)}{(x+y)/2}$. Moreover, because GUI Ripper does not include fuzz testing, we use coverage results from only the intelligent execution component for Playground. Using this metric, our measurements indicate Playground's intelligent component improves by a 31% in mean over GUI Ripper. We plot this difference against the number of applications in Figure 2.5. For applications on the positive side, Playground does better. Some applications lie on the negative side. This is likely because of non-determinism in applications because of which a run of GUI Ripper may be able to execute more code in an application than a different run of Playground. Such non deterministic behavior has been encountered earlier also [69].

2.9.2. Discussion

While event triggering is undoubtedly needed, it was not clear to us before the experiments how fuzz testing and intelligent execution would help and compare with each other. First, we found that the code coverage at simply launching the applications is only 16% while our automatic exploration techniques of fuzz testing and intelligent execution nearly double the code coverage. Second,

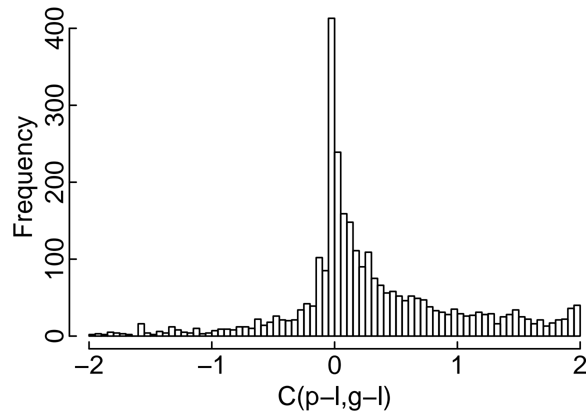


Figure 2.5. Percentage difference in code coverage between Playground and GUI Ripper

intelligent execution cannot work in cases that it does not model; this applies to all the custom-made widgets and, in the current implementation, to web-based GUI, which may also be embedded in apps and which is not handled currently (the process would be similar to handling normal GUI but in a different environment). In such cases, fuzz testing was found help, filling up the limitations of intelligent execution.

Intelligent execution was primarily useful in cases where user credentials or some meaningful information was required. In fact, for automatic login, we found that in several cases we had received emails on the email account we used for testing from several services. Playground had automatically created accounts with these services. In particular, we found emails from 34 different services. Some of these are popular social networking, cloud and media services such as Pandora, Dropbox, Last.fm, and Kik Messenger. Most of these related to account registrations while a few were received on supplying email address alone. We note that account registration for most applications is done through web sites. Playground currently cannot work with web pages. Moreover, many account registration routines also have captcha tests. However, once registered, Playground can easily use these credentials for subsequent navigation. A few situations were also

related to providing other meaningful inputs such as a city name or a zipcode. For example, the Weather.com app asks for this in the absence of consent to location data access. Exploration is quite stunted if this is not provided.

Intelligent execution is thus specially useful for complex apps, such as those for social networking. In these cases, fuzz testing is usually stuck at the beginning only due to need of login or similar things. It is however, usually after login only, that there is access to the user's databases, files, location and other sensor information.

2.10. Related Work

Dynamic Malware Analysis. Given we are trying to run applications and detect security and privacy breaches, our work naturally falls into the category traditionally known as dynamic malware analysis. For Android two works are quite comparable to our work. DroidScope [142] is a malware analysis framework for Android applications. It is however different from our work in that while we aim to detect malicious or unwanted functionality on a large scale (in thousands of apps), they aim at malware forensics, to provide accurate analysis of apps that are known to be malware. Their analysis does not provide automatic exploration and requires significant manual effort to understand the working of the malware.

Google Bouncer is a tool that screens applications uploaded to the Google Play market for malware. This tool appears to be similar to Playground in that it needs to provide automatic exploration and detection techniques. It is however a closed, proprietary tool and not much is known about it. Researchers [106, 137] have however found that it is poor at disguising techniques and many of the common identifiers may be used to identify the virtual environment.

Strider HoneyMonkey [136] loads webpages in the browser, automatically clicks dialog boxes to allow installation of any binary and then detect if it is malware. CWSandbox [138] and Botlab

[73] study malware behavior in monitored environments. All the above works have little or even no interaction with the malware executables being studied. Playground however is designed to work with highly interactive applications. These applications are different from the traditional malware in that the former's execution is primarily driven by interaction.

Driving Applications. Any dynamic program analysis approach may be classified as either a black-box or a white-box approach depending on whether it meaningfully uses the program code to do the analysis. For our automatic exploration, we decided to stick to the black-box (or a somewhat gray-box) approach which is far simpler than the white-box paradigms. Approaches like model checking [37] and symbolic and concolic execution [80, 129] would fall into the white-box space. We plan to include symbolic execution in the future in Playground. Zheng et al. [146] also propose a framework for automatic UI exploration of Android apps. It is a grey-box technique as some static analysis is involved. We can improve our approach by including similar static analysis to guide the dynamic exploration. However, as they also note static analysis is insufficient to analyze all aspects of the UI. Our black-box, yet sophisticated dynamic exploration techniques can help to cover such aspects.

GUI Testing. Automatic GUI testing has for long been an intriguing area in software engineering, specifically because of the complexity of event interactions that are possible. Much of the commercially available tools are directed towards capture-playback [21] or towards programmatic descriptions of input and output event sequences [15, 123]. These however do not provide completely automatic solutions to GUI testing. Our task at GUI exploration is obviously very different from what these tools can accomplish. Privacy Oracle [74] however uses capture-playback to its advantage for multiple runs along same paths on application GUI but with slightly perturbed inputs.

GUI testing is often accomplished as model based testing [18], involving coming up with a finite state machine model of the event space that the app provides and subsequent generation and execution of test cases based on that model. Given a model, automatic techniques may be used to come up with test cases [97, 114].

Memon et al. automatically deduce GUI models by exploring the GUI [95, 96]. We face a similar problem of automatically generating an abstract state machine by exploring the application UI. However, we model much more accurately window transitions without assuming a directed-acyclic-graph organization amongst windows (in Android, for example, cycles are possible). More importantly, Memon et al. do not provide abilities to fill contextual text input and do not talk about modules such as widget tracking and sequencing policies which we found crucial for black-box exploration. These advantages do show up in Section 2.9.

Hu and Neamtiu [70] have discover GUI bugs in Android applications. They fuzz applications and monitor the system logs to discover bugs. Playground can complement their work by driving applications automatically.

2.11. Conclusion and Future Work

In this chapter we proposed AppsPlayground, a tool automatic dynamic analysis of smartphone applications. We integrated a number of detection, exploration, and disguise techniques to come up with an effective analysis environment that may be used to evaluate Android applications on a large scale.

The future directions for Playground include including symbolic execution for systematic exploration of the applications' state space and to make Playground even more stealthy by enhancing the disguise techniques.

CHAPTER 3

DroidChameleon: Evaluating Android Anti-malware against Transformation Attacks

3.1. Introduction

Mobile computing devices such as smartphones and tablets are becoming increasingly popular. Unfortunately, this popularity attracts malware authors too. In reality, mobile malware has already become a serious concern. It has been reported that on Android, one of the most popular smartphone platforms [38], malware has constantly been on the rise and the platform is seen as "clearly today's target" [51, 94]. With the growth of malware, the platform has also seen an evolution of anti-malware tools, with a range of free and paid offerings now available in the official Android app market, Google Play.

We aim to evaluate the efficacy of anti-malware tools on Android in the face of various evasion techniques. For example, polymorphism is a common obfuscation technique that has been widely used by malware to evade detection tools by transforming a malware in different forms ("morphs") but with the same code. Metamorphism is another common technique that can mutate code so that it no longer remains the same but still has the same behavior. For ease of presentation, we use the term polymorphism to represent both obfuscation techniques. In addition, we use the term 'transformation' broadly, to refer to various polymorphic or metamorphic changes.

Polymorphic attacks have long been a plague for traditional desktop and server systems. While there exist earlier studies on the effectiveness of anti-malware tools on PCs [34], our domain of study is different in that we exclusively focus on mobile devices like smartphones that require

different ways for anti-malware design. Also, malware on mobile devices have recently escalated their evolution but the capabilities of existing anti-malware tools are largely not yet understood. In the meantime, there are warnings that Android malware will become more sophisticated, we will soon see polymorphic malware, and they will be able to quickly propagate from device to device using poisoned SMS messages and social network postings to infected links [58]. In fact, simple forms of polymorphic attacks have already been seen in the wild [135]. It is thus imperative for mobile security systems to have good defenses against polymorphic strains.

To evaluate existing anti-malware software, we develop a systematic framework called *Droid-Chameleon* with several common transformation techniques that may be used to transform Android applications automatically. Some of these transformations are highly specific to the Android platform only. Based on the framework, we pass known malware samples (from different families) through these transformations to generate new variants of malware, which are verified to possess the originals' malicious functionality. We use these variants to evaluate the effectiveness and robustness of popular anti-malware tools.

Our results on ten popular anti-malware products, some of which even claim resistance against malware transformations, show that all the anti-malware products used in our study have little protection against common transformation techniques. Many of them may even succumb to trivial transformations such as repacking that do not involve any code-level transformation. This is in contrast to the general understanding, also substantiated by reports from the industry [6, 14], that mobile anti-malware tools work quite well. Our evaluation dataset includes products that these reports claim to be perfect or nearly perfect. Our results also give insights about detection models used in existing anti-malware and their capabilities, thus shedding light on possible ways for their improvements. We hope that our findings work as a wake-up call and motivation for the community to improve the current state of mobile malware detection.

We emphasize that making judgment which anti-malware product is the best is a non-goal for this research. There are other important characteristics of anti-malware, such as the completeness of the signature database and resource consumption, that we do not evaluate. Additionally, security vendors typically package malware detection with other functionalities such as locating missing device or filtering spam SMS together in their offerings. Evaluating these functionalities remains beyond the scope of this worl.

To summarize, we the following contributions here.

- We systematically evaluate anti-malware products for Android regarding their resistance against various transformation techniques in known malware. For this purpose, we developed DroidChameleon, a systematic framework with various transformation techniques to facilitate anti-malware evaluation. Apart from general transformations, we also develop transformations that are specific to the Android platform.
- We have implemented a prototype of DroidChameleon and used it to evaluate ten popular anti-malware products for Android. Our findings show that all of them are vulnerable to common evasion techniques. Moreover, we find that 90% of the signatures studied do not require static analysis of bytecode.
- We studied the evolution of anti-malware tools over a period of one year. Our findings show that some anti-malware tools have tried to strengthen their signatures with a trend towards content-based signatures while previously they were evaded by trivial transformations non involving code-level changes. The improved signatures are however still shown to be easily evaded.
- Based on our evaluation results, we also explore possible ways to improve current anti-malware solutions. Specifically, we point out that Android eases advanced static analyses because much of the Android application code is high-level bytecodes rather than native

codes. Hence, anti-malware products could implement the already proposed semantics-based approaches for malware detection more easily for mobile platforms than for PCs where most applications are native binaries. Furthermore, certain platform support (in terms of offering higher privileges to anti-malware) can be enlisted to cope with advanced transformations.

The rest of this chapter is organized as follows. We present in Section 3.2 the necessary background and detail in Section 3.3 the DroidChameleon design. We then provide implementation details in Section 3.4 and summarize our malware and anti-malware data sets in Section 3.5. After that, we present our findings in Section 3.6, followed by a brief discussion in Section 3.7 on how to improve current anti-malware solutions. Finally, we examine related work in Section 3.8 and conclude in Section 3.9.

3.2. Background

Android is an operating system for mobile devices such as smartphones and tablets. It is based on the Linux kernel and provides a middleware implementing subsystems such as telephony, window management, management of communication with and between applications, managing application lifecycle, and so on. Third party applications run unprivileged on Android. The rest of this section will cover some background on the Android middleware and application fundamentals, application distribution, Android anti-malware, and signatures for malware detection.

3.2.1. Android Fundamentals

Applications are programmed primarily in Java though the programmers are allowed to do native programming via JNI (Java native interface). Instead of running Java bytecode, Android runs Dalvik bytecode, which is produced by the application build toolchain from Java bytecode. Dalvik

is a virtual machine designed to run in low-memory environments and is similar to the Java Virtual Machine (JVM) with the most notable difference being that it is register based (JVM is stack based). Most of the JVM concepts such as classes, class loaders, reflection, and so on are adopted as specified by the Java Language Specification in the Dalvik virtual machine. In Dalvik, instead of having multiple `.class` files as in the case of Java, all the classes are packed together in a single `.dex` (Dalvik Executable) file to minimize redundant strings and other constants. The dex file format keeps the Dalvik bytecode and specifies the organization of the various sections and items in the file. There are separate sections for keeping strings, class definitions, code items, and so on.

Android applications are made of four types of components, namely activities, services, broadcast receivers, and content providers. These application components are implemented as classes in application code and are declared in the AndroidManifest (see next paragraph). The Android middleware interacts with the application through these components. The reader is referred to the official Android Documentation for detail on these.

Android application packages are jar files[1] containing the application bytecode as a `classes.dex` file, any native code libraries, application resources such as images, config files and so on, and a manifest, called AndroidManifest. It is a binary XML file, which declares the application package name, a string that is supposed to be unique to an application, and the different components in the application. It also declares other things (such as application permissions) which are not so relevant to the present work. The AndroidManifest is written in human readable XML and is transformed to binary XML during application build.

Only digitally signed applications may be installed on an Android device. Application packages are signed similar to the signing of a jar file. Signing is only for the purpose of enabling better

[1] Java Archive format, which is really a zip file format

sharing among applications from the same developer and recognizing packages that come from the device vendor (such packages may have more privileges) and not verifying trust in the application. Signing keys are thus owned by individual developers and not by a central authority, and there is no chain of trust.

3.2.2. Android Anti-malware Solutions

With the proliferation of malware, there are now tens of both free and paid anti-malware products available in the official Android market. Many are from obscure developers while well-established, mainstream antivirus vendors offer others.

In order to get an insight on the workings of the anti-malware products, we briefly describe the necessary parts of the Android security model. Android achieves application sandboxing by means of Linux UIDs. Every application (with a few exceptions relating to how applications are signed) is given a separate UID and most of the application resources remain hidden from other UIDs.

Android anti-malware products are treated as ordinary third party applications and have no additional privileges over other applications. This is in contrast with the situation on traditional platforms such as Windows and Linux where antivirus applications run with administrator privileges. An important implication of this is that these anti-malware tools are mostly incapable of behavioral monitoring and do not have access to the private files of the application. The original application packages however remain intact and are readable by all applications. (Copy protected application packages are not readable by all applications but this feature is deprecated; paid applications are reportedly kept encrypted since Android 4.1.) These application packages may thus be used for static, signature-based malware detection. Moreover, Android provides a broadcast when a new application is installed. All the anti-malware applications we study have the ability to scan

applications automatically immediately following their installation, most likely by listening to this broadcast.

Android also provides a PackageManager API, which allows applications to retrieve all the installed packages. The API also allows getting the signing keys of these packages and the information stored in their AndroidManifest such as the package name, names of the components declared, the permissions declared and requested, and so on. Anti-malware applications have the opportunity to use information from this API as well for malware detection.

3.2.3. Malware Detection Signatures

While developing malware transformations, it is important to consider what kind of signatures anti-malware tools may use against malware. Signatures have traditionally been in the form of fixed strings and regular expressions. Anti-malware tools may also use chunks of code, an instruction sequence or API call sequence as signatures. Signatures that are more sophisticated require a deeper static analysis of the given sample. The fundamental techniques of such an analysis comprise data and control flow analysis. Analysis may be restricted within function boundaries (intra-procedural analysis) or may expand to cover multiple functions (inter-procedural analysis).

3.3. Framework Design

In this work, we focus on the evaluation of anti-malware products for Android. Specifically, we attempt to deduce the kind of signatures that these products use to detect malware and how resistant these signatures are against changes in the malware binaries. We generally use the term transformation to denote semantics preserving changes to a program. We next define transformations more specifically.

Let $\mathbf{P}$ be the set of all programs. A transformation is a mapping $\tau : \mathbf{P} \rightarrow \mathbf{P}$ that preserves the relevant semantics of the program. Note that we do not require all semantic behaviors to be preserved; we instead look for preserving only an interesting subset of behaviors of a given program. In case of malware, this interesting subset is the malicious behavior. For example, when a transformation corresponds to changing the package name of an application, the system logs about that application may show a different package name, but this behavior is not relevant. On the other hand, sending out a text message to a premium rate number without user consent is a relevant behavior when studying malware. Clearly, if two transformations preserve the relevant semantics, so will their composition.

In this work, we develop several different kinds of transformations that may be applied to malware samples while preserving their malicious behavior. Each malware sample undergoes one or more transformations and then passes through the anti-malware tools. The detection results are then collected and used to make deductions about the detection strengths of these anti-malware tools.

The transformation set in the DroidChameleon framework is comprehensive in the sense that we can expect to beat any static program analysis technique with these transformations. We also provide some Android-specific transformations (repacking and package renaming) which would give us important insights about the workings of Android anti-malware. Moreover, some transformations such as renaming identifiers and reflection do not apply to native code files typical to PCs. We classify our transformations as trivial (which do not require code level changes or changes to meta-data stored in AndroidManifest), those which result in variants that can still be detected by static analysis (DSA), and those which can render malware undetectable by static analysis (NSA). In the rest of this section, we describe the different kinds of transformations that we have in the DroidChameleon framework. Where appropriate we give examples, using original and

transformed code. Transformations for Dalvik bytecode are given in Smali (as in Listing 3.1), an intuitive assembly language for Dalvik bytecode and very similar to Jasmin assembly language for Java bytecode.

```
const-string v10, "profile"
const-string v11, "mount -o remount rw system\nexit\n"
invoke-static {v10, v11}, Lcom/android/root/Setting;->runRootCommand(Ljava/lang/String
    ;Ljava/lang/String;)Ljava/lang/String;
move-result-object v7
```

Listing 3.1. A code fragment from DroidDream malware

3.3.1. Trivial Transformations

Trivial transformations do not require code-level changes or changes to meta-data stored in AndroidManifest. These transformations are meant to defeat signatures based on whole files (or a part of file that changes simply by reorganizing file sections) or the key used to sign an application package. We have the following two transformations for this purpose.

Repacking. Recall that Android packages are signed jar files. These may be unzipped with the regular zip utilities and then repacked again with tools offered in the Android SDK. Once repacked, applications are signed with custom keys (the original developer keys are not available). Detection signatures that match the developer keys or a checksum of the entire application package are rendered ineffective by this transformation. Note that this transformation applies to Android applications only; there is no counterpart in general for Windows applications although the malware in the latter operating systems are known to use sophisticated packers for the purpose of evading anti-malware tools.

Disassembling and Reassembling. The compiled Dalvik bytecode in `classes.dex` of the application package may be disassembled and then reassembled back again. The various items in a dex file may be arranged or represented in different ways and thus a compiled program may be represented in more than one form. Signatures that match the whole `classes.dex` are beaten by this transformation. Signatures that depend on the order of different items in the dex file will also likely break with this transformation. Similar assembling/disassembling also applies to the resources in an Android package and to the conversion of AndroidManifest between binary and human readable formats.

3.3.2. Transformation Attacks Detectable by Static Analysis (DSA)

The application of DSA transformations does not break all types of static analysis. Specifically, forms of analysis that describe the semantics, such as data flows are still possible. Only simpler checks such as string matching or matching API calls may be thwarted. Except for certain forms (depending on the accuracy and detail of information needed) of data flow analysis and control flow analysis, we can expect other forms of detection described in Section 3.2.3 to be vulnerable to transformations described in this section.

3.3.2.1. Changing Package Name. Every application is identified by a package name unique to the application. This name is defined in the package's AndroidManifest. We change the package name in a given malicious application to another name. Package names of apps are concepts unique to Android and hence similar transformations do not exist in other systems.

3.3.2.2. Identifier Renaming. Similar to Java bytecode, Dalvik bytecode stores the names of classes, methods, and fields. It is possible to rename most of these identifiers without changing the semantics of the code. Constructors and methods that override super-class methods can however not be renamed. In general, such transformations apply only to source code or bytecode (which

preserve symbolic information) and not to native code. We note that several free obfuscation tools such as ProGuard [10] provide identifier renaming. Listing 3.2 presents an example transformation for code in Listing 3.1.

```
const-string v10, "profile"
const-string v11, "mount -o remount rw system\nexit\n"
invoke-static {v10, v11}, Lcom/hxbvgH/IWNcZs/jFAbKo;->axDnBL(Ljava/lang/String;Ljava/
    lang/String;)Ljava/lang/String;
move-result-object v7
```

Listing 3.2. Code in Listing 3.1 after identifier renaming

3.3.2.3. Data Encryption. The dex files contain all the strings and array data that have been used in the code. These strings and arrays may be used to develop signatures against malware. To beat such signatures we transform the dex file as follows. All the strings are stored in an encoded form, such as by the application of a simple Caesar cipher. Any access to an encoded string is immediately followed by a call to a routine for decoding the string. As an illustration, Listing 3.3 shows code in Listing 3.1, transformed by string encryption.

```
const-string v10, "qspgjmf"
invoke-static {v10}, Lcom/EncryptString;->applyCaesar(Ljava/lang/String;)Ljava/lang/
    String;
move-result-object v10
const-string v11, "npvou!.p!sfnpvou!sx!tztufn]ofyju]o"
invoke-static {v11}, Lcom/EncryptString;->applyCaesar(Ljava/lang/String;)Ljava/lang/
    String;
move-result-object v11
invoke-static {v10, v11}, Lcom/android/root/Setting;->runRootCommand(Ljava/lang/String
    ;Ljava/lang/String;)Ljava/lang/String;
move-result-object v7
```

Listing 3.3. Code in Listing 3.1 after string encryption. Strings are encoded with a Caesar cipher of shift +1.

The initialization data for arrays of primitive types is stored as bytes in the dex file. We encode these bytes using simple XOR cipher. Any operation to fill arrays with data is immediately followed by a call to a routine to decode the newly filled array.

3.3.2.4. Call Indirections. This transformation can be seen as a simple way to manipulate call graph of the application to defeat automatic matching. Given a method call, the call is converted to a call to a previously non-existing method that then calls the method in the original call. This can be done for all calls, those going out into framework libraries as well as those within the application code. This transformation may be seen as trivial function outlining (see function outlining below).

3.3.2.5. Code Reordering. Code reordering reorders the instructions in the methods of a program. This transformation targets detection schemes that rely on the order of the instructions, based on either the whole instructions, or part of the instructions such as opcodes. This transformation is accomplished by reordering the instructions and inserting `goto` instructions to preserve the runtime execution sequence of the instructions. We note that even though the Java language does not have a `goto` statement, the JVM and the Dalvik virtual machine both have the `goto` instruction. Since `goto` is not provided in the Java source language, a source level representation of the transformed program may not exist. Listing 3.4 shows an example reordering. Note that `move-result-*` must be the first instruction after a call to capture the return value.

```
goto :i_1
:i_3
invoke-static {v10, v11}, Lcom/android/root/Setting;->runRootCommand(Ljava/lang/String
    ;Ljava/lang/String;)Ljava/lang/String;
```

```
move-result-object v7
goto :i_4  # next instruction
:i_2
const-string v11, "mount -o remount rw system\nexit\n"
goto :i_3
:i_1
const-string v10, "profile"
goto :i_2
```

Listing 3.4. Code in Listing 3.1 reverse ordered

3.3.2.6. Junk Code Insertion. These transformations introduce code sequences that are executed but do not affect rest of the program. Detection based on analyzing instruction (or opcode) sequences may be defeated by junk code insertion. We propose two different kinds of transformations for this purpose: `nop` insertion, and arithmetic and branch insertion.

NOP insertion. This transformation simply inserts sequences of `nop` instructions in the code. It is easy to detect and undo.

Arithmetic and branch insertion. This transformation introduces junk arithmetic and branch instructions based on simple templates. The branch instructions have arbitrary branch offsets. The branch conditions are designed to be always false so that the branches are never actually taken. We assume that the value of these conditions (true or false) will be opaque to anti-malware tools being tested. Such obfuscation may create additional dependencies in control flow analysis. Listing 3.5 demonstrates some of the junk code that we generate. As in code reordering, we point out that there may not be a source level equivalent which compiles to the transformed program because branches are made to arbitrary offsets whereas control flow in Java is based on nested blocks (save the limited use of `break` and `continue`).

```
const/16 v0, 0x5
const/16 v1, 0x3
add-int v0, v0, v1
add-int v0, v0, v1
rem-int v0, v0, v1
if-lez v0, :junk_1
```

Listing 3.5. An example of a junk code fragment

3.3.2.7. Encrypting Payloads and Native Exploits. In Android, native code is usually made available as libraries accessed via JNI. However, some malware such as DroidDream also pack native code exploits meant to run from a command line in non-standard locations in the application package. All such files may be stored encrypted in the application package and be decrypted at runtime. Certain malware such as DroidDream also carry payload applications that are installed once the system has been compromised. These payloads may also be stored encrypted. We categorize payload and exploit encryption as DSA because signature based static detection is still possible based on the main application's bytecode. These are easily implemented and have been seen in practice as well (e.g., DroidKungFu malware uses encrypted exploit).

3.3.2.8. Function Outlining and Inlining. In function outlining, a function is broken down into several smaller functions. Function inlining involves replacing a function call with the entire function body. It is typically used by compilers for optimizing code related to short functions. The outlining refactoring has been proposed to eliminate duplicate code in programs [83]. However, outlining and inlining can be used for call graph obfuscation also. Outlining can also be used to impede all kinds of intra-procedural analyses. If a function is broken into sufficiently small chunks,

intra-procedural analysis will not be able to give any useful information. Interprocedural analysis is still possible though.

3.3.2.9. Other Simple Transformations. There are a few other transformations as well, specific to Android. Bytecode typically contains a lot of debug information, such as source file names, local and parameter variable names, and source line numbers. All this information may be stripped off. Another possible transformation is due to the nature of Android packages, which are zip files. Files archived in these zip files may be renamed. Finally, Android packages contain various resources apart from the `classes.dex` and AndroidManifest. All these resources may be renamed or modified appropriately.

3.3.2.10. Composite Transformations. Any of the above transformations may be combined with one another to generate stronger obfuscations. While compositions are not commutative, anti-malware detection results should be agnostic to the order of application of transformations in all cases discussed here.

3.3.3. Transformation Attacks Non-Detectable by Static Analysis (NSA)

These transformations can break all kinds of static analysis. Some encoding or encryption is typically required so that no static analysis scheme can infer parts of the code. Parts of the encryption keys may even be fetched remotely. In this scenario, interpreting or emulating the code (i.e., dynamic analysis) is still possible but static analysis becomes infeasible.

3.3.3.1. Reflection. Reflection is an easy way to obfuscate method calls. Reflection is the ability provided by certain programming languages allowing a program to introspect itself and change its behavior at runtime. In Java, the reflection API allows a program, among other things, to invoke a method by using the name of the methods. In reflection transformation, we convert every method call into a call to that method via reflection. This makes it difficult to analyze statically which

method is being called. A subsequent encryption of the method name can make it impossible for any static analysis to recover the call. Listing 3.6 illustrates code in Listing 3.1 after reflection transformation.

```
const-string v10, "profile"
const-string v11, "mount -o remount rw system\nexit\n"
const/4 v13, 0x2
new-array v14, v13, [Ljava/lang/Class;
new-array v15, v13, [Ljava/lang/Object;
const/4 v13, 0x0
const-class v12, Ljava/lang/String;
aput-object v12, v14, v13
aput-object v10, v15, v13
const/4 v13, 0x1
const-class v12, Ljava/lang/String;
aput-object v12, v14, v13
aput-object v11, v15, v13
const-string v13, "runRootCommand"
const-class v12, Lcom/android/root/Setting;
invoke-virtual {v12, v13, v14}, Ljava/lang/Class;->getMethod(Ljava/lang/String;[Ljava/
    lang/Class;)Ljava/lang/reflect/Method;
move-result-object v13
const/4 v16, 0x0
invoke-virtual {v13, v12, v15}, Ljava/lang/reflect/Method;->invoke(Ljava/lang/Object;[
    Ljava/lang/Object;)Ljava/lang/Object;
move-result-object v7
check-cast v7, Ljava/lang/String;
```

Listing 3.6. Listing 3.1 with method call by reflection

3.3.3.2. Bytecode Encryption. Code encryption tries to make the code unavailable for static analysis. The relevant piece of the application code is stored in an encrypted form and is decrypted at runtime via a decryption routine. Code encryption has long been used in polymorphic viruses; the only code available to signature based antivirus applications remains the decryption routine, which is typically obfuscated in different ways at each replication of the virus to evade detection. We discuss here code encryption alone; obfuscation of the decryption routine may be possible by other methods discussed above.

We accomplish bytecode encryption by moving most of the application in a separate dex file (packed as a jar) and storing it in the application package in an encrypted form. When one of the application components (such as an activity or a service) is created, it first calls a decryption routine that decrypts the dex file and loads it via a user defined class loader. In Android, the `DexClassLoader` provides the functionality to load arbitrary dex files. Following this operation, calls can be made into the code in the newly loaded dex file. Alternatively, one could define a custom class loader that loads classes from a custom file format, possibly containing encrypted classes. We note that classes which have been defined as components need to be available in `classes.dex` (one that is loaded by default) so that they are available to the Android middleware in the default class loader. These classes then act as wrappers for component classes that have been moved to other dex files.

3.4. Implementation

Apart from function outlining and inlining, we applied all other DroidChameleon transformations to the malware samples. We have implemented most of the transformations so that they may be applied automatically to the application. Automation implies that the malware authors can generate polymorphic malware at a very fast pace. Certain transformations such as native code

Table 3.1. Anti-malware products evaluated.

Vendor	Product	Package name	Version	# downloads
AVG	Antivirus Free	com.antivirus	3.1	50M-100M
Symantec	Norton Mobile Security	com.symantec.mobilesecurity	3.3.0.892	5M-10M
Lookout	Lookout Mobile Security	com.lookout	8.7.1-EDC6DFS	10M-50M
ESET	ESET Mobile Security	com.eset.ems	1.1.995.1221	500K-1M
Dr. Web	Dr. Web anti-virus Light	com.drweb	7.00.3	10M-50M
Kaspersky	Kaspersky Mobile Security	com.kms	9.36.28	1M-5M
Trend micro	Mobile Security Personal Ed.	com.trendmicro.tmmspersonal	2.6.2	100K-500K
ESTSoft	ALYac Android	com.estsoft.alyac	1.3.5.2	5M-10M
Zoner	Zoner Antivirus Free	com.zoner.android.antivirus	1.7.2	1M-5M
Webroot	Webroot Security & Antivirus	com.webroot.security	3.1.0.4547	500K-1M

Table 3.2. Malware samples used for testing anti-malware tools

Family	Package name	SHA-1 code	Date found	Remarks
DroidDream	com.droiddream. bowlingtime	72adcf43e5f945ca9 064b81dc0062007f0	03/2011	Root exploit
Geinimi	com.sgg.spp	1317d996682f4ae4cce6 0d90c43fe3e674f60c22	10/2011	Information exfiltration; bot-like capabilities
Fakeplayer	org.me.androidappli- cation1	1e993b0632d5bc6f0 0ee31e41dd316435d	08/2010	SMS trojan
Bgserv	com.android.vending. sectool.v1	bc2dedad0507a916604f 86167a9fa306939e2080	03/2011	Information exfiltration; bot-like capabilities; SMS trojan
BaseBridge	com.keji.unclear	508353d18cb9f5544 d1cf7ef8a0b6a5552	05/2011	Root exploit; SMS tro- jan packed as payload
Plankton	com.crazyapps.angry. birds.rio.unlocker	bee2661a4e4b347b5cd2 a58f7c4b17bcc3efd550	06/2011	Dynamic code loading

encryption are not possible to completely automate because one needs to know how native code files are being handled in the code.[2] Transformations that require modification of the AndroidManifest (rename packages and renaming components) have not been completely automated because we felt it was more convenient to modify manually the AndroidManifest for our study. Nevertheless, it is certainly possible to automate this as well. Finally, we did not automate bytecode encryption, although there are no technical barriers to doing that. However, we have implemented a proof-of-concept bytecode encryption transformation manually on existing malware.

[2]Native code stored in non standard locations is typically copied from the application package to the application directory by the application itself (possibly through an available Android API).

We utilize the Smali/Baksmali [12] and its companion tool Apktool [1] for our implementation. Apktool is able to unpack an application package, disassemble `classes.dex` into smali code and convert AndroidManifest to human readable form among other things. It can also assemble and repack a package. Most of the code transformations are applied to the smali assembly code, which is assembled later into dex code. Only method and field renaming was implemented directly on the dex code, yet using the underlying library for smali/baksmali. The assembling and disassembling transformation is implemented simply by decoding and building with Apktool. This has the effect of repacking, changing the order and representation of items in the `classes.dex` file, and changing the AndroidManifest (while preserving the semantics of it). All other transformations in our implementation (apart from repacking) make use of Apktool to unpack/repack application packages. Our overall implementation comprises about 1,100 lines of Python and Scala code.

We verified that our implementation of transformations do not modify the semantics of the programs. Specifically, we tested our transformations against several test cases and verified their correctness on two malware samples, DroidDream and Fakeplayer. In general, verifying correctness on actual malware is challenging because some of the original samples have turned non-functional owing to, for example, the remote server not responding, and because being able to detect all the malicious functionality requires a custom, appropriately monitored environment. Indeed, our original DroidDream sample would not work because it failed to get a reply from a remote server; we removed the functionality of contacting the remote server to confirm that the malicious functionality works as intended.

3.5. The Dataset

This section describes the anti-malware products and the malware samples we used for our study. We evaluated ten anti-malware tools, which are listed in Table 3.1. There are dozens of free

and paid anti-malware offerings for Android from various well-established anti-malware vendors as well as not-so-well-known developers. We selected the most popular products; in addition, we included Kaspersky and Trend Micro, which were then not very popular but are well established vendors in the security industry. We had to omit a couple of products in the most popular list because they would fail to identify many original, unmodified malware samples we tested. One of the tools, Dr. Web, actually claims that its detection algorithms are resilient to malware modifications.

Our malware set is summarized in Table 3.2. We used a few criteria for choosing malware samples. First, all the anti-malware tools being evaluated should detect the original samples. We here have a question of completeness of the signature set, which is an important evaluation metric for antivirus applications. In this work however, we do not focus on this question. Based on this criterion, we rejected Tapsnake, jSMSHider and a variant of Plankton. Second, the malware samples should be sufficiently old so that signatures against them are well stabilized. All the samples in our set were discovered in or before October 2011. All the samples are publicly available on Contagio Minidump [110].

Our malware set spans over multiple malware kinds. DroidDream [92] and BaseBridge [2] are malware with root exploits packed into benign applications. DroidDream tries to get root privileges using two different root exploits, rage against the cage, and exploid exploit. BaseBridge includes only one exploit, rage against the cage. If these exploits are successful, both DroidDream and BaseBridge install payload applications. Geinimi [4] is a trojan packed into benign applications. It communicates with remote C&C servers and exfiltrates user information. Fakeplayer [5], the first known malware on Android, sends SMS messages to premium numbers, thus costing money to the user. Bgserv [3] is a malware injected into Google's security tool to clean out DroidDream and distributed in third party application markets. It opens a backdoor on the device and exfiltrates

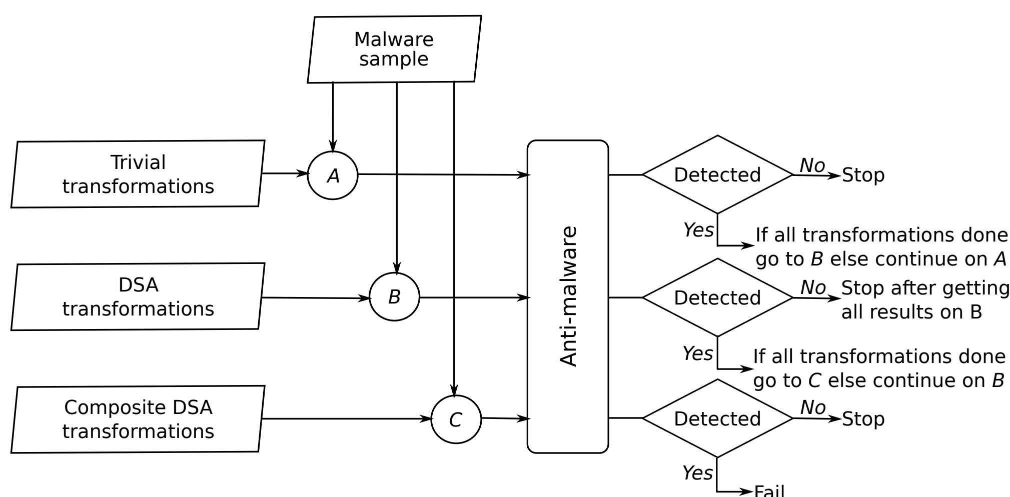


Figure 3.1. Evaluating anti-malware

user information. Plankton [9] is a malware family that loads classes from additional downloaded dex files to extend its capabilities dynamically.

3.6. Results

As has already been discussed, we transform malware samples using various techniques discussed in Section 3.3 and pass them through anti-malware tools we evaluate. We will now briefly describe our methodology and then discuss the findings of our study.

We describe our methodology through Figure 3.1 and through Tables 3.4 and 3.5, which depict the series of transformations applied to DroidDream and Fakeplayer samples and the detection results on various anti-malware tools. Empty cells in the tables indicate positive detection while cells with ‘x’ indicate that the corresponding anti-malware tool failed to detect the malware sample after the given transformations were applied to the sample. The tables reflect a general approach of our study. We begin testing with trivial transformations and then proceed with transformations that are more complex. Each transformation is applied to a malware sample (of course, some like

Table 3.3. Key to Tables 3.4, 3.5 and 3.6. Transformations coded with single letters are trivial transformations. All others are DSA. We did not need NSA transformations to thwart anti-malware tools.

Code	Technique
P	Repack
A	Dissassemble & assemble
RP	Rename package
EE	Encrypt native exploit or payload
RI	Rename identifiers
ED	Encrypt strings and array data
CR	Reorder code
CI	Call indirection
JN	Insert junk code
All transformations contain P	
All transformations except P contain A	

exploit encryption apply only in certain cases) and the transformed sample is passed through anti-malware. If detection breaks with trivial transformations, we stop.[3] Next, we apply all the DSA transformations. If detection still does not break, we apply combinations of DSA transformations. In general there is no well-defined order in which transformations should be applied (in some cases a heuristic works; for example, malware that include native exploits are likely to be detected based on those exploits). Fortunately, in our study, we did not need to apply combinations of more than two transformations to break detection. When applying combinations of transformations, we stopped when detection broke. We do not show the redundant combinations in the tables for the sake of conciseness. The last rows do not form part of our methodology; we construct them manually to show the set of transformations with which all anti-malware tools yield.

Our results with all the malware samples are summarized in Table 3.6. This table gives the minimal transformations necessary to evade detection for malware-anti-malware pairs. For example,

[3]All DSA and NSA transformations also result in trivial transformations because of involving disassembling, assembling and repacking. Hence, there is no use in proceeding further.

Table 3.4. DroidDream transformations and anti-malware failure. Please see Table 3.3 for key. ‘x’ indicates failure in detection.

	AVG	Symantec	Lookout	ESET	Dr. Web	Kaspersky	Trend M.	ESTSoft	Zoner	Webroot
P			x							
A			x						x	
RP	x		x					x	x	
EE			x						x	
RI		x	x						x	x
ED			x						x	
CR			x						x	
CI			x						x	
JN			x						x	
RI+EE		x	x	x					x	x
EE+ED			x			x			x	
EE+RF			x				x		x	
EE+CI			x		x				x	
RP+RI+EE +ED+RF+CI	x	x	x	x	x	x	x	x	x	x

Table 3.5. Fakeplayer transformations and anti-malware failure. Please see Table 3.3 for key. ‘x’ indicates failure in detection. EE transformation does not apply for lack of native exploit or payload in Fakeplayer.

	AVG	Symantec	Lookout	ESET	Dr. Web	Kaspersky	Trend Micro	ESTSoft	Zoner	Webroot
P										
A							x		x	
RP							x	x	x	x
RI				x		x	x		x	
ED							x		x	
CR							x		x	
CI					x		x		x	
JN							x		x	
RP+RI	x	x	x	x		x	x	x	x	x
RP+RI+CI	x	x	x	x	x	x	x	x	x	x

DroidDream requires both exploit encryption and call indirection to evade Dr. Web’s detection. These minimal transformations also give insight into what kind of detection signatures are being used. We next describe our key findings in the light of the detection results.

Finding 1 *All the studied anti-malware products are vulnerable to common transformations.* All the transformations appearing in Table 3.6 are easy to develop and apply, redefine only certain

Table 3.6. Evaluation summary. Please see Table 3.3 for key. ‘+’ indicates the composition of two transformations.

	DroidDream	Geinimi	Fakeplayer	Bgserv	BaseBridge	Plankton
AVG	RP	RI	RP + RI	RI	RI	RP + RI
Symantec	RI	RI	RP + RI	RI + ED	ED	P
Lookout	P	RI + ED	RP + RI	RI + ED	EE + ED	RI
ESET	RI + EE	ED	RI	RI	EE + ED	RI + ED
Dr. Web	EE + CI	CI	CI	CI	EE + CI	CI
Kaspersky	EE + ED	RI	RI	RI + ED	EE + ED	A
Trend M.	EE + RF	RI	A	A	EE + RF	A
ESTSoft	RP	RP	RP	RP	RP	RP
Zoner	A	RI	A	A	A	RI
Webroot	RI	RI	RP	RI	RP	RI

syntactic properties of the malware, and are common ways to transform malware. Transformations like identifier renaming and data encryption are easily available using free and commercial tools [10, 13]. Exploit and payload encryption is also easy to achieve. We point out that some of these transformations may already be seen in the wild in current malware. For example, Geinimi variants have encrypted strings [91]. Similarly, the DroidKungFu malware uses encrypted exploit code [8]; a similar transformation to DroidDream allows easy evasion across almost all the anti-malware tools we studied. No transformations just discussed thwart static analysis.

We found that only Dr. Web uses a somewhat more sophisticated algorithm for detection. Our findings indicate that the general detection scheme of Dr. Web is as follows. The set of method calls from every method is obtained. These sets are then used as signatures and the detection phase consists of matching these sets against sets obtained from the sample under test. We also tested Dr. Web against reflection transformation (not shown in the tables) and were able to evade it. This offers another confirmation that signatures are based on method calls. Furthermore, we also found (by limiting our transformations) that only framework API calls matter; calls within the application make no difference. It seems that the matching is somewhat fuzzy (requiring only a threshold percentage of matches) because we found on DroidDream and Fakeplayer that results

are positive even when a few classes are removed from the dex file. For these two families, we could create multiple minimal sets of classes that would result in positive detection. As mentioned earlier, Dr. Web indeed claims it has signatures that are resilient to malware modifications. It is difficult to say if the polymorphic resistance of these signatures is any stronger than other signatures depending on identifier names and string and data values. In particular, such signatures do not capture semantic properties of malware such as data and control flow. Our results aptly demonstrate the low resistance.

Finding 2 *At least 43% signatures are not based on code-level artifacts.* That is, these are based on file names, checksums (or binary sequences) or information easily obtained by the PackageManager API. We also found all AVG signatures to be derived from the content of AndroidManifest only (and hence that of the PackageManager API). In case of AVG, the signatures are based on application component classes or package names or both. Furthermore, this information is derived from AndroidManifest only. We confirmed this by placing a fake AndroidManifest in malware packages and assembling them with the rest of the package kept as it is. This AndroidManifest did not have any of the components or package names declared by the malware applications. The results were that detection was negative for all the malware samples.

Finding 3 *90% of signatures do not require static analysis of bytecode. Only one of ten anti-malware tools was found to be using static analysis.* Names of classes, methods, and fields, and all the strings and array data are stored in the `classes.dex` file as they are and hence can be obtained by content matching. The only signatures requiring static analysis of bytecode are those of Dr. Web because it extracts API calls made in various methods.

Finding 4 *Anti-malware tools have evolved towards content-based signatures over the past one year.* We studied compare our findings that we obtained in February 2012 (Table 3.7) to our present findings obtained in February 2013 (Table 3.6). Some of the anti-malware tools have

```
<manifest ... package="com.crazyapps.angry.birds.rio.unlocker" ... >
    <application android:label="@string/app_name" android:icon="@drawable/icon">
        <activity android:label="@string/app_name" android:name=
            ".AngryBirdsRioUnlocker" ... >
            :
        </activity>
        <service android:name="com.plankton.device.android.AndroidMDKProvider" ... />
    </application>
```

```
<manifest ... package="com.hDEWJu.oYlCvk.hFYkwc.FgDOHA.UPkmVF" ... >
    <application android:label="@string/app_name" android:icon="@drawable/icon">
        <activity android:label="@string/app_name" android:name=".LncHMH" ... >
            :
        </activity>
        <service android:name="com.rawJbA.DKPTQc.aaMYse.QUivSk" ... />
    </application>
```

Figure 3.2. An example evasion. Changes required in AndroidManifest of Plankton to evade AVG (original first and modified second; only relevant parts are shown with differences highlighted). No other changes are required. The application will not work though until the components are also renamed in the bytecode. We confirm that AVG's detection is based on the contents of AndroidManifest alone (see Finding 2).

Table 3.7. Summary of results from anti-malware tools downloaded in February 2012. Please see Table 3.3 for key. '+' indicates the composition of two transformations. Results that have changed since then are depicted in bold (see Table 3.6 for comparison).

	DroidDream	Geinimi	Fakeplayer	Bgserv	BaseBridge	Plankton
AVG	RP	RI	RP + RI	RI	RI	RP + RI
Symantec	**P**	RI	**RP**	**P**	**P**	P
Lookout	P	**ED**	**P**	**P**	EE + ED	RI
ESET	**EE**	ED	RI	RI	**EE**	**A**
Dr. Web	EE + CI	CI	CI	CI	EE + CI	CI
Kaspersky	**EE**	RI	RI	RI + ED	EE + ED	A
Trend M.	**EE**	RI	A	A	**EE**	A
ESTSoft	**P**	**P**	**P**	**P**	**P**	**P**
Zoner	A	**A**	A	A	A	**A**
Webroot	**RP**	**P**	RP	**P**	**P**	**RP**

changed considerably for the same malware samples. Last year, 45% of the signatures were evaded by trivial transformations, i.e., repacking and assembling/disassembling. Such signatures have virtually no resilience against polymorphism. Our present results show a marked decrease in this

fraction to 16%. We find that in all such cases where we see changes, anti-malware authors have moved to content-based matching, such as matching identifiers and strings.

Furthermore, for malware using native code exploits, many anti-malware tools previously matched on the native exploits and payloads alone. The situation has changed now as all of these additionally match on some content in the rest of the application as well. Although the changes in the signatures over the past one year may be seen as improvement, we point out that the new signatures still lack resilience against polymorphic malware as our results aptly demonstrate.

3.7. Defense against Transformation Attacks

In this section, we discuss how the current state of malware detection on Android may be improved. We identify how anti-malware tools should improve their detection techniques and that mobile platforms should provide special support to antimalware tools.

3.7.1. Semantics-based Malware Detection

We point out that owing to the use of bytecodes, which contain high-level structural information, analyses of Android applications becomes much simpler than those of native binaries. Hence, semantics-based detection schemes could prove especially helpful in the case of Android. For example, Christodorescu et al. [36] describe a technique for semantics based detection. Their algorithms are based on unifying nodes in a given program with nodes in a signature template (nodes may be understood as abstract instructions), while preserving def-use paths described in the template. Since this technique is based on data flows rather than a superficial property of the program such as certain strings or names of methods being defined or called, it is not vulnerable to any of the transformations (all of which are trivial or DSA) that show up in Table 3.6. Such a detection

scheme is arguably slower than current detection schemes but offers higher confidence in detection. This is just another instance of the traditional security-performance tradeoff. Christodorescu et al. had actually reported the running times to be in the order of a couple of minutes on their prototype and had suggested real performance is possible with an optimized implementation [36].

Semantics-based detection is quite challenging for native codes; their analyses frequently encounters issues such as missing information on function boundaries, pointer aliasing, and so on [68, 126]. Even disassembly of native binaries can be error prone [88, 128]. Stripped binaries pose even greater problems, which are not fully solved yet and current solutions for accurate disassembly require combination of static and dynamic techniques [102]. Bytecodes, on the other hand, preserve much of the source-level information, thus easing analysis. We therefore believe that anti-malware tools have greater incentive to implement semantic analysis techniques on Android bytecodes than they had for developing these for native code.

3.7.2. Support from Platform

Note that the use of code encryption and reflection (NSA transformations) can still defeat the above scheme. Code encryption does not leave visible code on which signatures can be developed (of course, the decryption routing may still be used for generating signatures). The use of reflection simply hides away the edges in the call graph. A sophisticated data flow analysis can still uncover those edges; however, if the method names used for reflective invocations are encrypted, these edges are rendered completely opaque to static analysis. Furthermore, it is possible to use function outlining to thwart any forms of intra-procedural analysis as well. Owing to these limitations, the use of dynamic monitoring is essential.

Recall that anti-malware tools in Android are unprivileged third party applications. This impedes many different kinds of dynamic monitoring that may enhance malware detection. We believe special platform support for anti-malware applications is essential to detect malware amongst stock Android applications. This can help malware detection in several ways. For example, a common way to break evasion by code encryption is to scan the memory at runtime. The Android runtime could provide all the classes loaded using user-defined class loaders to the anti-malware application. Once the classes are loaded, they are already decrypted and anti-malware tools can analyze them easily.

We note that providing privileges for dynamic monitoring to anti-malware applications would promote opportunities for malware to trick users to grant high privileges. This is again a trade-off. Anti-malware tools on PCs typically require high privileges and do useful work even though there are issues of fake antiviruses [98].

We note that Google recently introduced on-phone app verification [119], which checks the app checksum against a malware database upon installation. This however is not sufficient against polymorphic attacks each instance of a malicious app is unique. Google also performs offline app analysis for malware detection using its Bouncer service [90]. This is based on emulation (using virtual machines) of real phone environments. Such scanning by emulation however has its own problems, ranging from detection of a virtualized environment to the malicious activity not getting triggered in the limited time for which the emulation runs; Bouncer is no exception to this [106, 137]. We therefore believe offline emulation must be supplemented by strong static analysis or real-time dynamic monitoring.

3.8. Related Work

3.8.1. Evaluating Anti-malware Tools

AV-Test.org, an antivirus evaluation lab, rated anti-malware products for Android for the completeness of their detection [6, 14]. Our study is orthogonal to their study in that we evaluate how anti-malware products perform in detecting polymorphic variants of known malware. Most of the tools (9/10) we studied are rated as "very good" by them. This provides us reason to believe that the tools we did not study will not have any better resistance to polymorphism.

Zheng et al. [147] also studied the robustness of anti-malware against Android malware recently. They implement a subset of our transformations, use them generate several malware variants, and test these on VirusTotal, a webservice that tests submitted samples against over 40 antivirus products. Their results however only show the change in overall detection percentages as the transformations are applied. Our results are much stronger in that we can show that all anti-malware tools actually succumb for all malware samples tested. Moreover, we also deduce the weaknesses and strengths of some of the products. Finally, we abstained from using VirusTotal because we found that detection rates for some anti-malware (such as AVG and Dr. Web) are vastly different for the mobile version and the VirusTotal (perhaps desktop-based) version.

Christodorescu and Jha [34] conducted a study similar to ours on desktop anti-malware applications eight years ago. They also arrived at the conclusion that these applications have low resilience against malware obfuscation. Our study is based on Android anti-malware, and we include several aspects in our study that are unique to Android. Furthermore, our study dates after many research works (see below) on obfuscation resilient detection, and we would expect the proposed techniques to be readily integrated into new commercial products.

Finally, there are many works in the industry about the evaluation of desktop antivirus tools on metrics such as signature completeness, usability and so on [7, 124].

3.8.2. Obfuscation Techniques

Collberg et al. [39] review different types of obfuscations and classify them based on reverse engineering by a human and by automated tools, and the overhead added to the application. They propose many different obfuscations possible on Java (or Dalvik) code. Collberg et al. further propose sophisticated transformations such as modifying inheritance graphs and method cloning and implementation of opaque predicates (predicates whose outcome is difficult to arrive at while reverse engineering but is known to the obfuscator) to insert junk code [40, 41]. DroidChameleon provides only a few of the transformations proposed by them. Nonetheless, the set of transformations provided in DroidChameleon is comprehensive (together with the advanced transformations) in the sense that they can break typical static detection techniques used by anti-malware. As for opaque predicates, we use such techniques in our transformation for inserting junk code with the assumption that anti-malware tools will not be able to resolve conditions we use therein.

There are many tools that provide obfuscation for Java bytecode. Proguard [10] provides renaming of classes and class members. Other tools like Klassmaster [13] additionally provide flow obfuscation and string encryption. We provide much of these functionalities. While the goal of these tools is to evade manual reverse engineering, we aim at thwarting analysis by automatic tools.

3.8.3. Obfuscated Malware Detection

As already discussed, to deal with malware obfuscation, the detection techniques must be based on semantics rather than the syntax of the code. These detection techniques should therefore be based on data flow and control flow analyses of the samples under test. Christodorescu et al. [36] present

one such technique. Their algorithm is based on matching given samples against a template by unifying nodes in samples with nodes in the template while preserving def-use relationships. In subsequent work, Preda et al. [113] propose a semantics-based framework to prove properties about malware detectors. Kruegel et al. [84] tackle the problem of disassembling binaries that have been made hard to disassemble for malware analysis. Christodorescu et al. [35] and Fredrikson et al. [59] attempt to generate semantics based signatures by mining malicious behavior automatically. Kolbitsch et al. [82] also propose similar techniques. The last three works are for behavior-based detection and use different behavior representations such as data dependence graphs and information flows between system calls. Due to lower privileges for anti-malware tools on Android, these approaches cannot directly apply to these tools presently. Sequence alignment from bioinformatics [104, 132] has also been applied to malware detection and related problems [72, 139]. Further work is also there to compute statistical significance of scores given by these classical sequence alignment algorithms [16, 17]. It may be possible to adapt such techniques to detect transformed malware with high performance.

3.8.4. Smartphone Malware Research

With the growth of malware on smartphones, several research works have been done in this direction. DroidRanger [149] and Riskranker [64] use (mostly) static analysis to detect unknown malware from both known and unknown malware families. They identified several new malicious applications in the official Android market as well as alternative application markets. Peng et al. [111] investigate probabilistic models to rank risks for Android apps. Anti-malware authors may explore their approaches, which may serve as heuristics to raise malware suspicions. Crowdroid [30] uses crowd sourcing to collect system calls from applications running on mobile devices then uses clustering to identify malicious behavior. Such techniques cannot be currently used by

unprivileged third-party anti-malware applications on Android. Felt et al. [52] present a survey of smartphone malware. They present taxonomy of smartphone malware and explore the incentives to develop mobile device malware. Zhou et al. [148] provide another, more recent survey of Android malware. They study how well anti-malware tools detect malware samples found in the wild. The tools have good detection on some families, like Fakeplayer and Geinimi, but fail in our tests when the samples are transformed. Airmid [101] proposes new mobile infrastructure for malware mitigation. Apart from Android, they also explored malware on Symbian and iOS. Bose et al. [28] and Kim et al. [78] have used logical ordering of applications' actions and power consumption respectively to construct behavioral detection of Symbian malware. VirusMeter [89] also uses power consumption to catch misbehaving Symbian malware. It is still to be demonstrated if these techniques apply well to Android also. In a summary, none of the above works focuses on evaluating current mobile anti-malware solutions.

3.9. Conclusion

We evaluated ten anti-malware products on Android for their resilience against malware transformations. To facilitate this, we developed DroidChameleon, a systematic framework with various transformation techniques. Our findings show that all the anti-malware products evaluated are susceptible to common evasion techniques and may succumb to even trivial transformations not involving code-level changes. Finally, we explored possible ways in which the current situation may be improved and next-generation solutions may be developed.

CHAPTER 4

AutoCog: Measuring the Description-to-permission Fidelity in Android Applications

4.1. Introduction

Modern operating systems such as Android have promoted global ecosystems centered around large repositories or marketplaces of applications. Success of these platforms may in part be attributed to these marketplaces. Besides serving applications themselves, these marketplaces also host application metadata, such as descriptions, screenshots, ratings, reviews, and, in case of Android, permissions requested by the application, to assist users in making an informed decision before installing and using the applications. From the security perspective, applications may access users' private information and perform security-sensitive operations on the devices. With the application developers having no obvious trust relationships with the user, these metadata may help the users evaluate the risks in running these applications.

It is however generally known [56] that few users are discreet enough or have the professional knowledge to understand the security implications that may be derived from metadata. On Google Play, users are shown both the application descriptions and the permissions[1] declared by applications. An application's description describes the functionality of an application and should give an

[1]In Android, security-sensitive system APIs are guarded by permissions, which applications have to declare and which have to be approved at install-time.

idea about the permissions that would be requested by that application. We call this *description-to-permission fidelity*. For example, an application that describes itself as a social networking application will likely need permissions related to device's address book. A number of malware and privacy-invasive applications have been known to declare more permissions than their purported functionality warrants [52, 149].

With this belief that descriptions and permissions should generally correspond, we present *AutoCog*, a system that automatically identifies if the permissions declared by an application are consistent with its description. AutoCog has multi-fold uses.

- Application developers can use this tool to receive an early, automatic feedback on the quality of descriptions so that they improve the descriptions to better reflect the security-related aspects of the applications.
- End users may use this system to understand if an application is over-privileged and risky to use.
- Application markets can deploy this tool to bolster their overall trustworthiness.

The key challenge is to gather enough semantics from descriptions in natural language to reason about the permissions declared. We apply state-of-the-art techniques from natural language processing (NLP) for sentence structure analysis and computing semantic relatedness of natural language texts. We further develop our own learning-based algorithm to automatically derive a model that can be queried against with descriptions to get the expected permissions.

AutoCog is a substantial advancement over the previous state-of-the-art technique by Pandita et al. [108], who have also attempted to develop solutions with the same goals. Their tool called Whyper is primarily limited by the use of a fixed vocabulary derived from the platforms' API documents and the English synonyms of keywords there. Our investigations show that Whyper's

methodology is inherently limited regarding the following issues: (a) *Limited semantic information*: not all textual patterns associated with a permission can be extracted from API documents, e.g., <"*find*", "*branch atm*"> relate to location permissions and <"*scan*", "*barcode*"> relate to the permission for accessing the camera in our models but cannot conceivably be found from API documents; (b) *Lack of associated APIs*: certain permissions do not have associated APIs so that this methodology cannot be used; and (c) *Lack of automation*: it is not clear how the techniques could be automated. We have confirmed these limitations with Whyper's authors as well.

Our methodology is radically different from Whyper's as is evident from the following contributions of this chapter.

- *Relating descriptions and permissions.* We design a novel learning-based algorithm for modeling the relatedness of descriptions to permissions. Our algorithm correlates textual semantic entities (second contribution) to the declared permissions. It is noteworthy that the model is trained entirely from application descriptions and declared permissions over a large set of applications without depending on external data such as API documents, so that we do not have the problems of limited semantic information or lack of associated APIs from the very outset. Both training and classification are completely automatic.
- *Extracting semantics from descriptions.* We utilize state-of-the-art NLP techniques to automatically extract semantic information from descriptions. The key component for semantics-extraction in our design is Explicit Semantic Analysis (ESA), which leverages big corpuses like Wikipedia to create a large-scale semantics database, and which has been shown to be superior to dictionary-based synonyms and other methods [61] and is being increasingly adopted by numerous research and commercial endeavors. Such superior analysis further largely mitigates the problem of limited semantic information.

- *System prototype.* We design and implement an end-to-end tool called AutoCog to automatically extract relevant semantics from Android application descriptions and permissions to produce permission models. These models are used to measure description-to-permission fidelity: given an application description, a permission model outputs whether the permission is expected to be declared by that application. If the answer is yes, AutoCog further provides relevant parts of description that warrant the permission.

We further have the following evaluation and measurement highlights.

- *Evaluation.* Our evaluation on a set of 1,785 applications shows that AutoCog outperforms the previous work on detection performance and ability of generalization over various permissions by a large extent. AutoCog closely aligns with human readers in inferring the evaluated permissions from textual descriptions with an average precision of 92.6% and average recall of 92.0% as opposed to previous state-of-the-art precision and recall of 85.5% and 66.5% respectively.
- *Measurements.* Our findings on 45,811 applications using AutoCog show that the description-to-permissions fidelity is generally low on Google Play with only 9.1% of applications having permissions that can all be inferred from the descriptions. Moreover, we observe the negative correlation between fidelity and application popularity.

The remainder of this chapter is organized as follows. Section 4.2 gives further motivation of our work and presents a brief background and problem statement. Next we cover AutoCog design in detail in Section 4.3, followed by the implementation aspects in Section 4.4. Section 4.5 deals with the evaluation of AutoCog and introduces our measurement results. We have relevant discussion and related work in Sections 4.6 and 4.7. Finally, we conclude our work in Section 4.8.

4.2. Background and Problem statement

4.2.1. Background

Android introduces a sophisticated permission-based security model, whereby an application declares a list of permissions, which must be approved by the user at application installation. These permissions guard specific functionalities on the device, including some security and privacy-sensitive APIs such as access contacts.

Modern operating systems such as Android, iOS, and Windows 8 have brought about the advent of big, centralized application stores that host third-party applications for users to view and install. Google Play, the official application store for Android, hosts both free and paid applications together with a variety of metadata including the title and description, reviews, ratings, and so on. Additionally, it also provides the user with the ability to study the permissions requested by an application.

4.2.2. Problem Statement

The application descriptions on Google Play are a means for the developers to communicate the application functionality to the users. From the security and privacy standpoint, these descriptions should thus indicate the reasons for the permissions requested by an application, either explicitly or implicitly[2]. We call it *fidelity* of descriptions to permissions.

As stated in Section 4.1, Android applications often have little in their descriptions to indicate to the users why they need the permissions declared. Specifically, there is frequently a gap between the access of the sensitive device APIs by the applications and their stated functionality. This may not always be out of malicious intent; however users are known to be concerned about the use of

[2]By implicit, we mean that the need for permission is evident from stated functionality.

sensitive permissions [54]. Moreover, Felt et al. [56] show that few users are careful enough or able to understand the security implications derived from the metadata. In this work we thus look into the problem of *automatically assessing the fidelity of the application descriptions with respect to the permissions.*

Detection of malicious smartphone applications is possible through static/run-time analysis of binaries [48, 69, 142]. However, the techniques to evaluate whether application oversteps the user expectation are still lacking. Our tool can assist the users and other entities in the Android ecosystem assess whether the descriptions are faithful to the permissions requested. AutoCog may be used by users or developers individually or deployed at application markets such as Google Play. It may automatically alert the end users if an application requests more permissions than required for the stated functionalities. The tool can provide useful feedback about the shortcomings of the descriptions to the developers and further help bolster the overall trustworthiness of the mobile ecosystem by being deployed at the markets.

As for automatically measuring description-to-permission fidelity, we need to deal with two concepts: (a), the *description semantics*, which relates to the meaning of the description, and (b), the *permission semantics*, which relates to the functionality provided (or protected) by the permission. The challenges in solving our problem therefore lie in:

- *Inferring description semantics*: Same meaning may be conveyed in a vast diversity of natural language text. For example, the noun phrases "*contact list*", "*address book*", and "*friends*" share similar semantic meaning.
- *Correlating description semantics with permission semantics*: A number of functionalities described may map to the same permission. For example, the permission to access user location might be expressed with the texts "*enable navigation*", "*display map*", and

"*find restaurant nearby*". The need for permission to write to external disk can be implied as "*save photo*" or "*download ringtone*".

In AutoCog, we consider the decision version of the problem stated above: *given a description and a permission, does the description warrant the declaration of the permission?* If AutoCog answers yes, it provides the sentences that warrant the permission, thus assisting users in reasoning about the requested permission. As a complete system, AutoCog solves this decision problem for each permission declared.

Whyper [108] is a previous work with goals similar to ours. Whyper correlates the description and permission semantics by extracting natural language keywords from an external source, Android API documents. Since APIs and permissions can be related together [20], the intuition is that keywords and patterns expressed in the API documentation will also be found in the application descriptions and are therefore adequate in representing the respective permissions. Based on our investigation, the methodology has the following fundamental limitations:

- *Limited semantic information*: the API documents are limited in the functionality they describe and so Whyper cannot cover a complete set of semantic patterns correlated with permissions. For example, in our findings, the pattern <"*deposit*", "*check*"> is related to the permission CAMERA with high confidence but cannot be extracted from API documents. The mobile banking applications, such as Bank of America[3], support depositing by snapping its photo with the device's camera. Analysis on this issue in detail will be shown in Section 4.5.2.
- *Lack of associated APIs*: certain sensitive permissions do not have any associated APIs. RECEIVE_BOOT_COMPLETED is one example [20]. It is thus not possible to generate the correlated textual pattern set with the API documents.

[3] https://play.google.com/store/apps/details?id=com.infonow.bofa

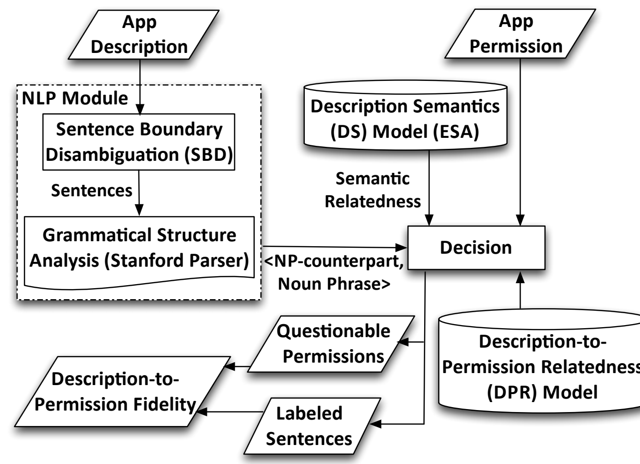


Figure 4.1. Overall architecture of AutoCog

- *Lack of automation*: Whyper's extraction of patterns from API documents involved manual selection to preserve the quality of patterns; what policies could be used to automate this process in a systematic manner is an open question.

Our learning-based approach automatically discovers a set of textual patterns correlated with permissions from the descriptions of a rich set of applications, hence enabling our description-to-permission relatedness model to achieve a complete coverage over the natural language texts with great diversity. Besides, the training process works directly on descriptions. So we easily overcome the limitations of the previous work as stated above.

4.3. System Design

Figure 4.1 gives an architectural overview of AutoCog. The description of the application is first processed by the NLP module, which disambiguates sentence boundaries and analyzes each sentence for grammatical structure. The output of the NLP module is then passed in together with the application permissions into the decision module, which, based on models of description semantics and description-to-permission relatedness outputs the questionable permissions that are

not warranted from the description and the sentences from which the other permissions may be inferred. These outputs together provide description–to-permission fidelity. This section provides a detailed design of each of the modules and the models that constitute AutoCog.

4.3.1. NLP Module

The goal of the NLP module is to identify specific constructs in the description such as noun and verb phrases and understand relationship among them. Use of such related constructs alleviates the shortcomings of simple keyword-based analysis. The NLP module consists of two components, sentence boundary disambiguation and grammatical structure analysis.

4.3.1.1. Sentence boundary disambiguation (SBD). The whole description is split into sentences for subsequent sentence structure analysis [81, 122]. Characters such as ".", ":", "-", and some others like "*", "♠", "◇" that may start bullet points are treated as sentence separators. Regular expressions are used to annotate email addresses, URLs, IP addresses, Phone numbers, decimal numbers, abbreviations, and ellipses, which interfere with SBD as they contain the sentence separator characters.

4.3.1.2. Grammatical structure analysis. We leverage Stanford Parser [133] to identify the grammatical structure of sentences. While our design depends on constructs provided by the Stanford Parser, it is conceivable that other NLP parsers could be used as well.

We first use the Stanford Parser to output *typed dependencies*, which are semantic hierarchies of sentences, i.e., how different parts of sentences depend on each other. As illustrated in Figure 4.2, the dependencies are triplets: *name of the relation*, *governor* and *dependent*. *Part of Speech (PoS)* tagging additionally assigns a part-of-speech tag to each word; for example, a verb, a noun, or an adjective. The results are fed into *phrase parsing* provided by Stanford Parser to break sentences

into phrases, which could be noun phrases, verb phrases or other kinds of phrases. We obtain a hierarchy of marked phrases and tagged words for each sentence.

The governor-dependent pair provides the knowledge of logic relationship between various parts of sentence, which provides the guideline of our ontology modeling. The concept of ontology is a description of things that exist and how they relate to each other. In our experience, we find the following ontologies, which are governor-dependent pairs based on noun phrase, to be most suitable for our purposes.

- Logical dependency between verb phrase and noun phrase potentially implies the actions of applications performing on the system resources. For example, the pairs <"scan", "barcode"> and <"record", "voice"> reveal the use of permissions camera and recording.
- Logical dependency between noun phrases is likely to show the functionalities mapped with permissions. For instance, users may interpret the pairs <"scanner", "barcode"> and <"note", "voice"> as using camera and microphone.
- Noun phrase with own relationship (possessive, such as "your", followed by resource names) is recognized as requesting permissions. For example, the RECORD_AUDIO and CAMERA permissions could be revealed by the pairs <"own", "voice"> and <"your", "camera"> .

We extract all the noun phrases in the leaf nodes of the hierarchical tree output from grammatical structure analysis. For each noun phrase, we record all the verb phrases and noun phrases that are its ancestors or siblings of its ancestors. We also record the possessive, if the noun phrase itself contains the own relationship. For the sake of simplicity, we call the extracted verb phrases, noun phrases, and possessives as *np-counterpart* for the target noun phrase. The noun-phrase based

```
Sample Sentence: "Search for a place near your location as
well as on our interactive maps"
(ROOT
 (S
  (VP (VB Search - 1)
   (PP
    (PP (IN for - 2)
     (NP
      (NP (DT a - 3) (NN place - 4))
      (PP (IN near - 5)
       (NP (PRP$ your - 6) (NN location - 7)))))
    (CONJP (RB as - 8) (RB well - 9) (IN as - 10))
    (PP (IN on - 11)
     (NP (PRP$ our - 12) (JJ interactive - 13) (NNS maps - 14)))))))
det(place-4, a-3)
prep_for(Search-1, place-4)
poss(location-7, your-6)
prep_near(place-4, location-7)
poss(maps-14, our-12)
amod(maps-14, interactive-13)
prep_on(Search-1, maps-14)
```

Figure 4.2. Example output of Stanford Parser

governor-dependent pairs obtained signify direct or indirect dependency. The example hierarchy tree for sentence “*Search for a place near your location as well as on our interactive maps*” is shown in Figure 4.2 with the pairs extracted: <“search”, “interactive map”>, <“our”, “interactive map”>, <“search”, “place”>, <“search”, “location”>, <“place”, “location”>, and <“your”, “location”>.

We process these pairs to remove stopwords and named entities. *Stopwords* are common words that cannot provide much semantic information in our context, *e.g.*, “the”, “and”, “which”, and so on. Named entities include names of persons, places, companies, and so on. These also do not communicate security-relevant information in our context. To filter out named entities, we employ *named entity recognition*, a well-researched NLP topic, also implemented in Stanford Parser. The remaining words are normalized by lowercasing and *lemmatization* [45]. Example normalizations include “better” $\rightarrow$ “good” and “computers” $\rightarrow$ “computer”.

4.3.2. Description Semantics (DS) Model

The goal here is to understand the meaning of a natural language description, i.e., how different words and phrases in a vocabulary relate to each other. Similarly meaning natural language descriptions can differ vastly; so such an analysis is necessary. Our model is constructed using *Explicit Semantic Analysis (ESA)*, the state of the art for computing semantic relatedness of texts [61]. The model is used directly by the decision module and also for training the description-to-permission relatedness model discussed in Section 4.3.3.

ESA is an algorithm to measure the semantic relatedness between two pieces of text. It leverages big document corpuses such as Wikipedia as its knowledge base and constructs a vector representation of text. In ESA, each (Wiki) article is called a concept, and transformed into a weighted vector of words within the article. As processing an input article, ESA computes the relatedness of the input to every concept, i.e. projects the input article into the concept space, by the common words between them. In NLP and information retrieval applications, ESA computes the relatedness of two input articles using the cosine distance between the two projected vectors.

We choose ESA because it has been shown to outperform other known algorithms for computing semantic relatedness such as WordNet and latent semantic analysis [61]. We offer intuitive reasons of out-performance over WordNet as this has been used in Whyper. First, WordNet-based methods are inherently limited to individual words, and adoption for comparing longer text requires an extra level of sophistication [99]. Second, considering words in context allows ESA to perform word sense disambiguation. Using WordNet cannot achieve disambiguation, since information about synsets (sets of synonyms) is limited to a few words; while in ESA, concepts are associated with huge amounts of text. Finally, even for individual words, ESA offers a much more detailed and quantitative representation of semantics. It maps the meaning of words/phrases to a

Table 4.1. Distribution of noun phrase patterns used

Pattern	#Noun Phrase (Percentage %)
Noun	1,120,850 (52.37 %)
Noun + Noun	414,614 (19.37 %)
Adjective + Noun	278,785 (13.03 %)
Total	1,814,249 (84.77 %)

Note: The above patterns are from a total of 2,140,225 noun phrases extracted from 37,845 applications.

weighted combination of concepts, while mapping a word in WordNet amounts to simple lookup, without any weight.

4.3.3. Description-to-Permission Relatedness (DPR) Model

Description-to-permission relatedness (DPR) model is a decisive factor in enhancing the accuracy of AutoCog. We design a learning-based algorithm by analyzing the descriptions and permissions of a large dataset of applications to measure how closely a noun-phrase based governor-dependent pair is related to a permission. The flowchart for building the DPR model is shown in Figure 4.3. We first leverage ESA to group the noun phrases with similar semantics. Next, for each permission, we produce a list of noun phrases whose occurrence in descriptions is positively related to the declaration of that permission. Such phrases may potentially reveal the need for the given permission. In the third stage, we further enhance the results by adding in the np-counterparts (of the noun-phrase based governor-dependent pairs) and keeping only the pairs whose occurrence statistically correlates with the declaration of the given permission.

4.3.3.1. Grouping Noun Phrases. A noun phrase contains a noun possibly together with adjectives, adverbs, etc. During the learning phase, since analyzing long phrases is not efficient, we consider phrases of only three patterns: single noun, two nouns, and noun following adjective

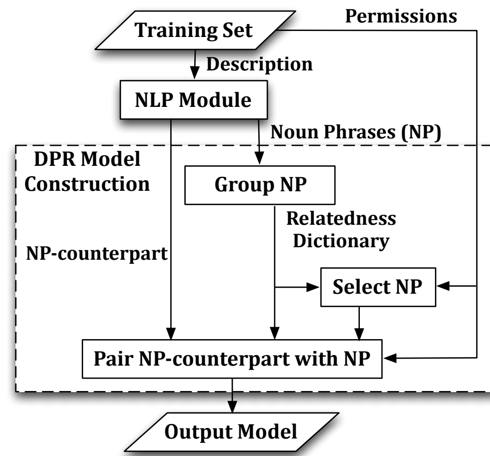


Figure 4.3. Flowchart of description-to-permission relatedness (DPR) model construction

(Table 4.1). In our dataset of 37,845 applications, these patterns account for 85% of the 302,739 distinct noun phrases. We further note that we focus on these restricted patterns only during DPR model construction; all noun phrases are considered in the decision module of AutoCog, which checks whether the description of application indicates a given permission. The DS model, which is also employed during decision-making, can match longer patterns with similarly meaning noun phrases grouped here. Hence the negative effect of such simplification is negligible.

We construct a *semantic relatedness score matrix* leveraging DS model with ESA. Each cell in the matrix depicts the semantic relatedness score between a pair of noun phrases. Define the *frequency* of noun phrase to be the number of applications whose descriptions contain the noun phrase. As constructing the semantic relatedness score matrix has quadratic runtime, it is not scalable and efficient. We filter out noun phrases with low frequencies from this matrix, as the small number of samples cannot provide enough confidence in our frequency-based measurement. If a low-frequency phrase is similar to a high-frequency phrase, our decision process will not be affected as the decision module employs DS model. We choose a threshold; only phrases with

frequency above 15 are used to construct the matrix. The number of such phrases in our dataset is 9,428 (3.11%).

Using the semantic relatedness score matrix, we create a *relatedness dictionary*, which maps a given noun phrase to a list of noun phrases, all of which have a semantic relatedness score higher than the threshold θ_g. The interpretation is that the given noun phrase may be grouped with its list of noun phrases as far as semantics is concerned. Our implementation takes θ_g to be 0.67. The lists also record the corresponding semantic relatedness scores for later use. A sample dictionary entry of the noun phrase "*map*" is:

$<$ "map", [("map", 1.00), ("map view", 0.96), ("interactive map", 0.89),...] $>$

4.3.3.2. Selecting Noun Phrases Correlated With Permissions. Whether a certain noun phrase is related to a permission is learnt statistically from our dataset. If a permission $perm$ and a noun phrase np appear together (i.e., $perm$ in permission declarations and np in the description) in a high number of applications, it implies a close relationship between the two. This is however not trivial; some noun phrases (*e.g.*, "game" and "application") may occur more frequently than others, biasing such calculations. Moreover, some noun phrases may actually be related to permissions but statistical techniques may not correlate them if they occur together in only a few cases in the dataset. The latter is partially resolved by leveraging the relatedness dictionary from the previous step. Based on existing data mining techniques [107], we design a quality evaluation method that (a) is not biased to frequently occurring noun phrases, and (b) takes into account semantic relatedness between noun phrases to improve the statics of meaningful noun phrases that occurs less than often. For the permission $perm$ and the noun phrase np, the variables in the learning algorithm are defined as:

$\mathbf{MP(perm, np)}$: An application declares $perm$. Either np or any noun phrase with the semantic

relatedness score to np above the threshold θ_g is found in the description. This variable will increase by 1, if np is in the description, or it will increase by the maximal relatedness score of the noun phrase(s) related to np.

$\mathbf{MMP(perm, np)}$: An application does NOT declare $perm$. Either np or any noun phrase with the semantic relatedness score to np above the threshold θ_g is found in the description. This variable will increase by 1, if np is in the description, or it will increase by the maximal relatedness score of the noun phrase(s) related to np.

$\mathbf{PR(perm, np)}$: The ratio of $MP(perm, np)$ to the sum of $MP(perm, np)$ and $MMP(perm, np)$:

$$PR(perm, np) = \frac{MP(perm, np)}{MP(perm, np) + MMP(perm, np)}.$$

$\mathbf{AVGPR(perm)}$: The percentage of all the applications in our training set that request $perm$.

$\mathbf{INCPR(perm, np)}$: This variable measures the increment of the probability that $perm$ is requested with the presence of np or its related noun phrases given the unconditional probability as the baseline:

$$INCPR(perm, np) = \frac{PR(perm, np) - AVGPR(perm)}{AVGPR(perm)}.$$

$\mathbf{MMNP(perm, np)}$: An application declares $perm$. This variable will increase by 1, if none of np and noun phrases related to it in the Relatedness Dictionary are found in the description.

$\mathbf{NPR(perm, np)}$: The ratio of $MP(perm, np)$ to the sum of $MP(perm, np)$ and $MMNP(perm, np)$:

$$NPR(perm, np) = \frac{MP(perm, np)}{MP(perm, np) + MMNP(perm, np)}.$$

$\mathbf{AVGNP(np)}$: Expectation on the probability that one description contains np or related noun phrases over the training set. Assume the total number of applications is M. This variable is

expressed as:

$$AVGNP(np) = \frac{\Sigma_{i=1}^{i=M} \lambda_i}{M},$$

where λ_i equals 1, if np is in the description of the *i-th* application. Or it equals to the maximal semantic relatedness score of its related noun phrase(s) found in description. If neither np nor noun phrases related to it in the Relatedness Dictionary are found, λ_i = 0.

$\mathbf{INCNP(perm, np)}$: This variable measures the growth on the probability that one description includes np or the related noun phrases with the declaration of $perm$ given expectation as the baseline:

$$INCNP(perm, np) = \frac{NPR(perm, np) - AVGNP(np)}{AVGNP(np)}.$$

Semantic relatedness score is taken as weight in the calculations of variables $MP(perm, np)$ and $MMP(perm, np)$, which groups the related noun phrases and resolves the minor case issue. We should note that $INCPR(perm, np)$ and $INCNP(perm, np)$ evaluate the quality of np by the growth of the probabilities that $perm$ is declared and np (or noun phrases related to np) is detected in description with the average level as baseline. This design largely mitigates the negative effect caused by the intrinsic frequency of noun phrase. To roundly evaluate the quality of np of describing $perm$, we define the $Q(perm, np)$, which is the harmonic mean of $INCPR(perm, np)$ and $INCNP(perm, np)$:

$$Q(perm, np) = \frac{2 \cdot INCPR(perm, np) \cdot INCNP(perm, np)}{INCPR(perm, np) + INCNP(perm, np)}.$$

np with negative values of $INCPR$ or $INCNP$ is discarded as it shows no relevance to $perm$. Each permission has a list of noun phrases, arranged in descending order by the quality value. The *top-k* noun phrases are selected for the permission. We set k=500 after checking the distribution of quality value for each permission. It is able to give a relatively complete semantic coverage of

the permission. Increasing the threshold k excessively would enlarge the number of noun-phrase based governor-dependent pairs in the DPR model. So it would reduce the efficiency of AutoCog in matching the semantic meaning for the incoming descriptions.

4.3.3.3. Pair np-counterpart with Noun Phrase. By following the procedure presented in Section 4.3.3.2, we can find a list of noun phrases closely related to each permission. However, simply matching the permission with noun phrase alone fails to explore the context and semantic dependencies, which increases false positives. Although a noun phrase related to "*map*" is detected in the example sentence below, it does not reveal any location permission.

"*Retrieve Running Apps" permission is required because, if the user is not looking at the widget actively (for e.g. he might using another app like Google Maps)*"

To resolve this problem, we leverage Stanford Parser to get the knowledge of context and typed dependencies. For each selected noun phrase np, we denote as $G(np)$ the set of noun phrases that have semantic relatedness scores with np higher than θ_g. Given a sentence in description, our mechanism identifies any noun phrase $np' \in G(np)$ and records each np-counterpart nc (recall that np-counterpart was defined as a collective term for verb phrases, noun phrases, and possessives for the target noun phrase), which has direct/indirect relation with np'. For each noun-phrase based governor-dependent pair $< nc, np >$, let the total number of descriptions where the pair $< nc, np' >$ is detected be SP. In the SP applications, let the number of application requesting the permission is tc. We keep only those pairs for which (1) $tc/SP > Pre_T$, (2) $SP > Fre_T$, where Pre_T and Fre_T are configurable thresholds. Thus we maintain the precision and the number of samples large enough to yield statistical results with confidence.

4.3.4. Decision

In DPR model, each permission has a list of related pairs of np-counterpart nc_{dpr} and noun phrase np_{dpr}, which reveal the security features of the permission. For an input application whose description has to be checked, the NLP module extracts the pairs of np-counterpart nc_{new} and noun phrase np_{new} in each sentence. We leverage the DS model to measure the semantic relatedness score $RelScore(txt_A, txt_B)$ between the two texts txt_A and txt_B. The sentence is identified as revealing the permission, if $< nc_{new}, np_{new} >$ is matched with a pair $< nc_{dpr}, np_{dpr} >$ by fulfilling the conduction:

$$RelScore(nc_{new}, nc_{dpr}) > \Upsilon,$$

$$RelScore(np_{new}, np_{dpr}) > \Theta.$$

Here, Υ and Θ are the thresholds of the semantic relatedness score for np-counterparts and noun phrases. The sentences indicating permissions will be annotated. Besides, AutoCog finds all the questionable permissions, which are not warranted in description.

4.4. Implementation

NLP Module: We use the NLTK library in Python and regular expression matching to implement the SBD. NLTK is also used for removing stopwords and normalizing words using lemmatization based on WordNet. Stanford Named Entity Recognizer is used for removing named entities.

DS and DPR Models: Noun phrases are classified by frequency. High-frequency noun phrases are grouped based on semantic relatedness score by utilizing the library `esalib`[4]. This library is the only currently maintained, open-source implementation of ESA that we could find. Our training algorithm on descriptions and permissions of large-scale applications selects the semantic patterns, which strongly correlate with the target permission by leveraging the frequency-based

[4]`https://github.com/ticcky/esalib`

measurement and ESA. Our current implementation pairs np-counterpart of length one (noun, verb, and possessive) with noun phrases. The np-counterpart could be easily extended to multiple words, possibly with a few considerations about maximum phrase length, and so on.

Overall, We implement AutoCog with over 7,000 lines of code in Python and 500 lines of code in Java.

4.5. Evaluation

We first describe our dataset and methodology for collecting sensitive permissions. Then, AutoCog's accuracy is evaluated by comparing with Whyper [108]. Finally, we discuss our measurements, which investigate the overall trustworthiness of market and the correlation between description-to-permission fidelity and application popularity.

4.5.1. Permission Selection and Dataset

The Android APIs have over a hundred permissions. However, some permissions such as the permission VIBRATE, which enables vibrating the device, may not be as sensitive as, for example, the permission RECORD_AUDIO, which enables accessing the microphone input. It is not so useful to identify permissions that are not considered sensitive. The question to ask then is, *what permissions are the users most concerned about from the security/privacy perspective?*

Felt et al. [54] surveyed 3,115 smartphone users about 99 risks and asked the participants to rate how upset they would be if a given risk occurred. We infer 36 Android platform permissions from the risks with highest user concerns. Since we focus here on third-party applications, we first remove from this list the Signature/System permissions, which are granted only to applications that are signed with the device manufacturer's certificate. Seven permissions were removed as a result. The 29 remaining permissions are arranged in descending order by the percentage of applications

requesting it in our dataset, which is collected randomly. We select the top 14 permissions in our evaluation, because the ground-truth of our evaluation relies on readers to identify whether sentences in application description imply sensitive permissions; the consequent human efforts make it difficult to review large number of descriptions.

We collected the declared permissions and descriptions of 37,845 Android applications from Google Play in August 2013 for the purpose of training the DPR model and evaluate AutoCog's accuracy. The permissions that constitute the subject of our study can be divided into 3 categories according to the abilities that they entail: (1) accessing user privacy, (2) costing money, and (3) other sensitive functionalities. Applications request the permissions to access privacy may leak users' personal information such as location to third parties without being awared. Permissions costing money, such as CALL_PHONE, may be exploited resulting in financial loss to the users. Other sensitive permissions may change settings, start applications on boot, thus possibly wasting phone's battery, and so on. In Table 4.2, we list the number and percentage of applications declaring each permission in our dataset.

We also parsed the metadata of another 45,811 Android applications from Google Play in May 2014 for our measurements, which assess the description-to-permission fidelity of large-scale applications in Google Play and investigate the correlation between description-to-permission fidelity with application popularity. The metadata include the following features: *category of application, developer of application, number of installations, average rating, number of ratings, descriptions and declared permissions of application.*

Table 4.2. Permissions used in evaluation over 37,845 applications

Permission	#App (Percentage %)
WRITE_EXTERNAL_STORAGE	30384 (80.29 %)
ACCESS_FINE_LOCATION	16239 (42.91 %)
ACCESS_COARSE_LOCATION	15987 (42.24 %)
GET_ACCOUNTS	12271 (32.42 %)
RECEIVE_BOOT_COMPLETED	9912 (26.19 %)
CAMERA	6537 (17.27 %)
GET_TASKS	6214 (16.42 %)
READ_CONTACTS	5185 (13.70 %)
RECORD_AUDIO	4202 (11.10 %)
CALL_PHONE	3130 (8.27 %)
WRITE_SETTINGS	3056 (8.07 %)
READ_CALL_LOG	2870 (7.58 %)
WRITE_CONTACTS	2176 (5.74 %)
READ_CALENDAR	817 (2.16 %)

4.5.2. Accuracy Evaluation

4.5.2.1. Methodology. Whyper studied three permissions: READ_CALENDAR, READ_CONTACTS, and RECORD_AUDIO; Their public results are directly utilized[5] as the ground-truth. The validation set contains around 200 applications for each of the three permissions, where each sentence in the descriptions is identified if revealing the permission by human readers. Moreover, to assess AutoCog's ability of generalization over other permissions in Table 4.2, we further randomly select 150 applications requiring each one (except the three permissions previously evaluated in public results of Whyper) as the validation set. For each permission, the complementary set of the validation set is used as the training set to construct the DPR model, which ensures that the validation set is independent of the training set. To get the results of Whyper on other permissions, we leverage the output of PScout [20] and manually extract the semantic pattern set from Android

[5] `https://sites.google.com/site/whypermission/`

API document[6] following the method presented by Pandita et al. [108]. Whyper's methodology does not work for some permissions such as RECEIVE_BOOT_COMPLETED as they do not have any associated API. To ensure the correctness of our understanding of Whyper's methodology, we contacted Whyper's authors and confirmed our understanding and conclusions. We also tested the system over the applications in their public results and get exactly the same output as those published, further validating the system deployment (source code is released publicly).

Regarding the ground-truth of other permissions that we extend to, we invite 3 participants who are not authors of this chapter to read the description and label each sentence as whether or not it suggests the target permission. The description will be classified as "good" when at least two human readers could infer the permission by one sentence in that, or it will be labeled as "bad". Column G_d" in Table 4.3 is the percentage of "good" descriptions for applications requesting each sensitive permission. The percentage values of "good" descriptions for the 3 permissions GET_TASKS, CALL_PHONE, and READ_CALL_LOG are lower than 10%. We call these permissions rarely described well in descriptions, *hidden permissions*. The scarcity of qualified descriptions leads to the lack of correlated semantic patterns. It would hinder the measurement of description-to-permission fidelity. After removing the 3 hidden permissions, our evaluation focuses on the other 11 permissions.

In training the DPR model, the two thresholds Pre_T and Fre_T balance the performance on precision and coverage of AutoCog. The settings in Table 4.3 depend on the percentage of applications requesting the permission in the training set. For a permission with fewer positive samples (application requires that permission), each pair of np-counterpart and noun phrase related to it

[6] http://pscout.csl.toronto.edu/download.php?file=results/jellybean_publishedapimapping

Table 4.3. Statistics and settings for evaluation

Permission	Fre_T	Pre_T	G_d %
WRITE_EXTERNAL_STORAGE	9	0.87	38.7
ACCESS_FINE_LOCATION	6	0.85	40.7
ACCESS_COARSE_LOCATION	5	0.8	35.3
GET_ACCOUNTS	4	0.8	26.0
RECEIVE_BOOT_COMPLETED	5	0.85	37.3
CAMERA	3	0.8	48.7
GET_TASKS	3	0.9	2.0
READ_CONTACTS*	3	0.8	56.8
RECORD_AUDIO*	3	0.8	64.0
CALL_PHONE	2	0.8	10.0
WRITE_SETTINGS	2	0.85	44.7
READ_CALL_LOG	3	0.95	6.0
WRITE_CONTACTS	2	0.9	42.0
READ_CALENDAR*	1	0.85	43.6

* Sampled by around 200 applications, others by 150 applications
Hidden permissions are grayed

tends to be less dominant in amount, we adjust Fre_T accordingly to maintain the performance on recall. We keep Pre_T high across permissions, which aims at enhancing the precision of detection.

Within the process of deciding if each application description in valuation set warrants permissions, we set the two thresholds Υ=0.8 and Θ=0.67 by empirically finding the best values for them. Low threshold reduces the performance on precision and increasing the threshold excessively causes the increment on false negatives. We set up the threshold Θ lower than Υ, because noun phrases has more diversity in patterns than np-counterparts; phrases containing various numbers of words organized in different orders may express the similar meaning.

Our objective is to assess how closely the decision made by AutoCog on the declaration of permission approaches human recognition given a description. The number of true positives, false positives, false negatives, and true negatives are denoted as TP: the system correctly identifies a description as revealing the permission, FP: the system incorrectly identifies a description as

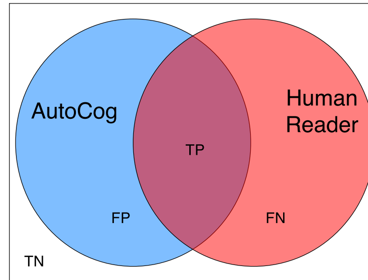


Figure 4.4. Interpretation of metrics in evaluation

revealing the permission, FN: the system incorrectly identifies a description as not revealing the permission, and TN: the system correctly identifies a description as not revealing the permission. Interpretation of the metrics is shown in Figure 4.4. Intersection of decisions made by AutoCog and human is true positive. Difference sets between decisions made by AutoCog and human are false positive and false negative, respectively. Complementary set of the union of decisions made by AutoCog and human is true negative. Values of $precision$, $recall$, $F\text{-}score$, and $accuracy$ represent the degree to which AutoCog matches human reader's recognition in inferring permission by description.

$$Precision = \frac{TP}{TP + FP},$$
$$Recall = \frac{TP}{TP + FN},$$
$$F\text{-}score = \frac{2 \cdot Precision \cdot Recall}{Precision + Recall},$$
$$Accuracy = \frac{TP + TN}{TP + FP + TN + FN}.$$

4.5.2.2. Results. Results of our evaluation are given in Table 4.4. AutoCog matches human in inferring 11 permissions with the average precision, recall, F-score, and accuracy as 92.6%,

Table 4.4. Results of evaluation

System	Permission	TP	FP	FN	TN	$Prec$ (%)	Rec (%)	F (%)	$Accu$ (%)
AutoCog	WRITE_EXTERNAL_STORAGE	53	6	5	86	89.8	91.4	90.6	92.7
	ACCESS_FINE_LOCATION	57	3	4	86	95.0	93.4	94.2	95.3
	ACCESS_COARSE_LOCATION	49	1	4	96	98.0	92.5	95.1	96.7
	GET_ACCOUNTS	34	4	5	107	89.5	87.2	88.3	94.0
	RECEIVE_BOOT_COMPLETED	51	6	5	88	89.5	91.1	90.3	92.7
	CAMERA	67	7	6	70	90.5	91.8	91.2	91.3
	READ_CONTACTS	99	5	9	77	95.2	91.7	93.4	92.6
	RECORD_AUDIO	117	10	11	62	92.1	91.4	91.8	89.5
	WRITE_SETTINGS	65	7	2	76	90.3	97.0	93.5	94.0
	WRITE_CONTACTS	57	4	6	83	93.4	90.5	91.9	93.3
	READ_CALENDAR	79	5	6	105	94.0	92.9	93.5	94.4
	Total	728	58	63	936	92.6	92.0	92.3	93.2
Whyper	WRITE_EXTERNAL_STORAGE	11	8	47	84	57.9	19.0	28.6	63.3
	ACCESS_FINE_LOCATION	31	1	30	88	96.9	50.8	66.7	79.3
	ACCESS_COARSE_LOCATION	28	1	25	96	96.6	52.8	68.3	82.7
	GET_ACCOUNTS	9	2	30	109	81.8	23.1	36.0	78.7
	RECEIVE_BOOT_COMPLETED					*Failed to get results*			
	CAMERA	26	4	47	73	86.7	35.6	50.5	66.0
	READ_CONTACTS	89	9	19	73	90.8	82.4	86.4	85.3
	RECORD_AUDIO	105	10	23	62	91.3	82.0	86.4	83.5
	WRITE_SETTINGS	59	24	8	59	71.1	88.1	78.7	78.7
	WRITE_CONTACTS	53	9	10	78	85.5	84.1	84.8	87.3
	READ_CALENDAR	78	15	7	95	83.9	91.8	87.6	88.7
	Total	489	83	246	817	85.5	66.5	74.8	79.9

92.0%, 92.3%, and 93.2%. As discussed before, Whyper fails to get results for permission RECEIVE_BOOT_COMPLETED. For the remaining 10 permissions, Whyper achieves the average precision, recall, F-score, and accuracy as 85.5%, 66.5%, 74.8%, and 79.9%.

Across the permissions evaluated, the least precision and recall of AutoCog are 89.5% and 87.2%. Even for the cases with low percentage of "good" descriptions and low number of positive samples (permissions GET_ACCOUNT and READ_CALENDAR), our learning-based algorithm and employment of ESA could still get the DPR model aligning with user's recognition well. Whyper could only infer 5 permissions from description (last 5 in Table 4.4) with both the values of precision and recall higher than 70%. For these permissions, the API documents provide a relatively complete and accurate semantic pattern set. The example patterns such as <"*scan*","*wifi*">,

<“*enable*”,“*bluetooth*”>, and <“*set*”,“*sound*”> could be extracted from the API document of the permission WRITE_SETTINGS. However, Whyper does not perform well on the other 5 permissions. Our understanding is that the patterns extracted from API documents in these cases are very limited to cover the natural-language descriptions with great diversity. For example, the APIs mapped with the permission to write to external storage are related only to download management. Many intuitive patterns, such as <“*save*”, “*sd card*”>, <“*transfer*”, “*file*”>, <“*store*”, “*photo*”> cannot be found in its API document. It is the same with <“*scan*”, “*barcode*”>, <“*record*”, “*video*”> for camera permission, <“*integrate*”, “*facebook*”> (in-app login) for permission to get user’s accounts, and <“*find*”, “*branch*”>, <“*locate*”, “*gas station*”> for location permissions. Given Whyper’s big variance of performance and our investigation on its source of textual pattern set, we find that suitability of API document to generate a complete and accurate set of patterns varies with permissions due to the limited semantic information in APIs. AutoCog relies on large number of descriptions in training, which would not be restricted by the limited semantic information issue and has stronger ability of generalization over permissions.

Whether or not the API documents are suitable for the evaluated permissions, we note that AutoCog outperforms Whyper on both $precision$ and $recall$. Next we discuss several case studies to thoroughly analyze the benefits and limitations of our design.

AutoCog *TP*/Whyper *FN*: The advantage of AutoCog over Whyper on false negative rate (or recall) is caused by: (1) the difference in the fundamental method to find semantic patterns related to permissions, (2) we include the logical dependency between noun phrases as extra ontology. Whyper is limited by the use of a fixed and limited set of vocabularies derived from the Android API documents and their synonyms. Our correlation of permission with noun-phrase based governor-dependent pair is based on clustering results from a large application dataset, which is much richer than that extracted from API documents. Below are 3 examples:

"*Filter by contact, in/out SMS*"

"*Blow into the mic to extinguish the flame like a real candle*"

"*5 calendar views (day, week, month, year, list)*"

The first sentence describes the function of backing up SMS by selected contact. The second sentence reveals a semantic action of blowing into the microphone. The last sentence introduces one calendar application, which provides various views. In our DPR model, the noun-phrase based governor-dependent pairs <*filter, contact*>, <*blow, mic*>, and <*view, calendar*> are found to be correlated to the 3 permissions, READ_CONTACTS, RECORD_AUDIO, and READ_CALENDAR. While the semantic information for the first two sentences cannot be found by leveraging the API documents. For the last one, Whyper could only detect it, as "*view*" and "*calendar*" are tagged with *verb* and *noun*, respectively (both of them are tagged as *noun* here).

AutoCog *TN*/Whyper *FP*: One major reason for this difference in detection is that Whyper is not able to accurately explore the meaning of noun phrase with multiple words. Below is one example:

"*Saving event attendance status now works on Android 4.0*"

The sentence tells nothing about requiring the permission to access calendar. However, Whyper incorrectly labels it as revealing the permission READ_CALENDAR, because it parses resource name "event" and maps it with action "save". AutoCog differentiates the two phrases " event attendance status" and "event" by using ESA and effectively filters the interference in DPR model training and decision-making.

AutoCog *FN*/Whyper *TP*: This difference is caused by the fact that some semantic patterns implying permissions are not included in the DPR model. Below is one example:

"*Ability to navigate to a Contact if that Contact has address*"

Whyper detects the word "contact" as resource name and maps it with the verb "navigate". The sentence is thus identified as revealing the permission to read the address book. However, no

noun-phrase based governor-dependent pair in our DPR model could be mapped to the permission sentence above, because the pair <*navigate*, *contact*> is not dominant in the training process. The DPR model might not be knowledgeable enough to completely cover the semantic patterns related to the permission. However, the coverage could be enhanced as the size of training set increases.

AutoCog *FP*/Whyper *TN*: In the training process, some semantic patterns, which do not directly describe the reason for requesting the permission in the perspective of user expectation, are selected in the frequency-based measurement. One example is given as:

"*Set recordings as ringtone*"

From this sentence, user could customize her/his ringtone with recording, but it does not directly imply the functionality of recording sound. Our model assigns a high relatedness score between <*set, recording*> and RECORD_AUDIO due to quite a few training samples with related keywords and this permission together. Such cases are due to the fundamental gap between machine learning and human cognition.

AutoCog and Whyper both leverage Stanford Parser [133] to get the tagged words and hierarchal dependency tree. The major cause of the common erroneous detection of two systems (*FP*, *FN*) is the incorrect parsing of sentence by underlying NLP infrastructure, which has been well stated by Pandita et al. [108]. Thus, we would not discuss it in detail given the page limit. As the research in the field of NLP advances underlying NLP infrastructure, the number of such errors will be reduced.

We further list some representative semantic patterns in Table 4.5, which are found to be closely correlated by our DPR model to the permissions evaluated.

Apart from the accuracy of detection, the runtime latency is a key metric in the practical deployment of AutoCog. We select 500 applications requiring each permission and assess the runtime

Table 4.5. Example semantic patterns

Permission	Semantic Patterns
WRITE_EXTERNAL_STORAGE	<*delete*, *audio file*>; <*convert*, *file format*>; <*download*, *ringtone*>
ACCESS_FINE_LOCATION	<*display*, *map*>; <*find*, *branch atm*>; <*your*, *location*>
ACCESS_COARSE_LOCATION	<*set*, *gps navigation*>; <*remember*, *location*>; <*inform*, *local traffic*>
GET_ACCOUNTS	<*manage*, *account*>; <*integrate*, *facebook*>; <*support*, *single sign-on*>
RECEIVE_BOOT_COMPLETED	<*change*, *hd wallpaper*>; <*display*, *notification*>; <*allow*, *news alert*>
CAMERA	<*deposit*, *check*>; <*scanner*, *barcode*>; <*snap*, *photo*>
READ_CONTACTS	<*block*, *text message*>; <*beat*, *facebook friend*>; <*backup*, *contact*>
RECORD_AUDIO	<*send*, *voice message*>; <*note*, *voice*>; <*blow*, *microphone*>
WRITE_SETTINGS	<*set*, *ringtone*>; <*customize*, *alarm*>; <*enable*, *flight mode*>
WRITE_CONTACTS	<*wipe*, *contact list*>; <*secure*, *text message*>; <*merge*, *specific contact*>
READ_CALENDAR	<*optimize*, *time*>; <*synchronize*, *calendar*>; <*schedule*, *appointment*>

latency of our system in measuring the description-to-permission fidelity. *AutoCog achieves the latency less than 4.5s for all the 11 permissions.*

4.5.3. Measurement Results

Our measurements begin with assessing the overall trustworthiness of application market, which is depicted by the distribution of questionable permissions. We utilize AutoCog with the DPR model trained in the accuracy evaluation to analyze 45,811 applications. The training set and dataset for measurements are thus disjoint. The histogram for distribution of questionable permissions is illustrated in Figure 4.5. Only 9.1% of applications are clear of questionable permissions. Moreover, we measure and observe the negative Spearman correlation [76] between the number of questionable permissions of one application by a specific developer with the total number of applications published by that developer (with $r = -0.405$, $p < 0.001$). A possible explanation is that developer publishing more applications are more experienced and likely to be a development team in a company, who is more standardized and better regulated at developing and deploying its mobile software. The above results reflect the severity of the permission-to-description fidelity issue: application publishers, especially the new or personal developer, generally fail to completely cover

Table 4.6. Correlation between application popularity and the number of questionable permissions and permissions requested. All values are statistically significant with $p < 0.001$

	Correlation with application popularity		
Permission Type	# Installs	# Ratings	avg rating
$\#P_q$	-0.106	-0.105	-0.110
$\#P$	0.044	0.050	0.044

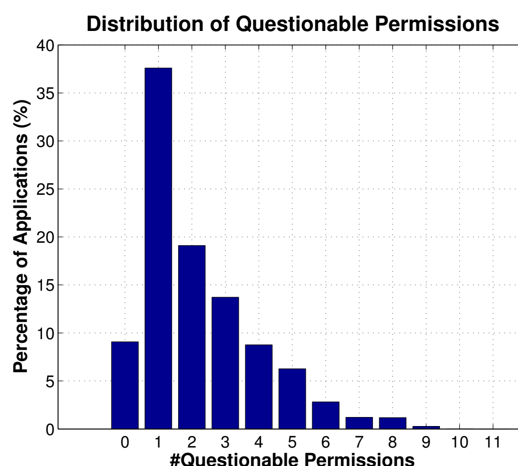


Figure 4.5. Histogram for distribution of questionable permissions

all the sensitive permissions. The deployment of AutoCog could thus assist developers produce applications with high description-to-permissions fidelity.

We further investigate the correlation between description-to-permission fidelity and application popularity. Application popularity reveals the developers' benefit and users' attitude towards the application, which thus plays a key role in the interaction between users and developers. In our measurements, application popularity is interpreted by the following features: number of installations, number of ratings, average ratings. Thus, we measure the Spearman correlation coefficient between these three features with the number of questionable permissions ($\#P_q$) and the number of permissions ($\#P$) requested by application, respectively. Table 4.6 shows that there is a weak positive correlation between application popularity and the number of permissions requested, which

is consistent with the results by other researchers [32, 55]. It is because that rich functionality of application which implies the need of more permissions is the main feature to drive application popularity.

However, we also find the *weak negative correlation* between the number of questionable permissions and the popularity of application. We should note that all the measured results achieve a p-value less than 0.001, which means the statistical significance. We have the following two guesses. First, for the negative correlation, there are a small part of users who are discreet enough or have the professional knowledge to fully understand the security aspects of application metadata [56]. They expect to get permission-related information from the description. Thus the low description-to-permission fidelity negatively affects their decisions of application installation, application assessment, and interest in applications. Secondly, such correlation is weak because most average users cannot tell the questionable permissions based on the description without a tool like AutoCog. Although we could only confirm correlation but not causation here, we expect that wide adoption of AutoCog will help average users to be more security conscious.

4.6. Discussion

AutoCog measures the description-to-permission fidelity by finding relationships between textual patterns in the descriptions and the permissions. Because of the state-of-the-art techniques used and the new modeling techniques developed, AutoCog achieves good accuracy. Still, AutoCog does have limitations because of the approach it uses and the current implementation.

The models learnt in AutoCog are examples of unsupervised learning, which has the drawback of picking relationships that may not actually exist directly. If a noun phrase appears frequently with a permission, the DPR model will learn that they are actually related. For example, if many

antivirus applications use the permission GET_TASKS, the "antivirus" noun may become associated with this permission even if there is no direct relationship between the two. From another perspective though, one could argue that this is even better because AutoCog may be able to extract implicit relationships that human readers may easily miss. Anecdotally, for applications with permission GET_TASKS in our experiments, even if human readers could find only 2% of applications whose descriptions reveal that permission, AutoCog finds 18% of such applications.

For the implementation of AutoCog, we could possibly improve the accuracy by including longer noun phrases and np-counterparts. It is an efficiency-accuracy tradeoff. The evaluation of AutoCog also had some limitations. Manual reading is subjective and the results may be biased. However, given that our readers have a technical background, they may be able to discover many implicit relationships that average users ignore, thus putting up greater challenges for AutoCog. Given that whether a description implies a permission itself is subjective and is consequently lack of ground-truth, manual labeling is the best we can do here.

Malicious developers may provide wrong descriptions to evade this approach. But it will be much easier for even average users to find such mismatch between the app's description and its functionality. And given that most apps are not malicious, such attacks will not affect the training of AutoCog.

4.7. Related Work

NLP has been widely used in the security area. Potharaju et al. [112] propose an approach to analyzing natural language text in network tickets to infer the problem symptoms and resolution actions. Some efforts have focused on automating mining of network failures from syslogs [116] and network logs [85]. Compared with the network tickets and logs, descriptions of applications have much more complex structures and diverse contents, which largely increases the difficulties

of ontology modeling. For example, the developer could choose to use either complete sentences or enumeration lists in description; introduction and contact of company may be included for commercial purpose. There are also approaches using a mix of NLP and learning algorithm to infer specifications from API descriptions, code comments, and formal requirement documents [109]. The methods proposed in these papers require meta-information from source code. Our design only needs the natural language text of descriptions, which is not constrained by the availability of source code and meta-information.

The permission system in Android security framework manages the access of third-party applications to privacy- and security-relevant parts of API. Many previous studies analyze the permission system and resolve the overprivilege issue [20,53], confused deputy [33,44,57] and collusion attack [29]. Moreover, some studies also investigate the effectiveness of permission model [55,77]. Some researchers have alluded to lack of correlation between permissions and descriptions [25]; however, even if permissions and descriptions do not correlate, our solution can bring an improvement to the current situation. Lin et al. [86] utilize crowdsourcing collect users expectations of the permissions required by application and Han et al. [66] propose a text mining-based similarity measure method to obtain similar security polices among Android applications, which are both complimentary to our work. While the static/run-time analysis of binaries and programming language analysis enable these approaches to detect overprivilege and confused deputy attack, the end user does not have knowledge about why the permission is requested or tools to assess whether applications overstep user expectation. Our system analyzes the descriptions of applications that the end user has direct and easy access to and labels the sentences revealing sensitive permissions, which enables users to know the reason for declaring the permission in the semantic level.

The most relevant work is Whyper [108], which is the only previous work to our knowledge on bridging the gap between what user expects an application to do and what it really does. Our

automatic learning-based approach works directly on large-scale descriptions to select noun-phrase based governor-dependent pairs related to each permission. Thus we would not come across the limitations of Whyper discussed in Section 4.2.2.

4.8. Conclusion

In this chapter, we propose the system AutoCog that measures the description-to-permissions fidelity in Android, i.e., whether the permissions requested by Android applications match or can be inferred from the applications' descriptions. The use of a novel learning-based algorithm and advanced NLP techniques allows us to mine relationships between textual patterns and permissions. AutoCog outperforms previous work on both performance of detection and ability of generalization over permissions by a large extent. In inferring eleven permissions by description, our system achieves the average precision of 92.6% and the average recall of 92.0% as compared to previous state-of-the-art 85.5% and 66.5%. Our measurements show a generally weak description-to-permissions fidelity on the Google Play store.

CHAPTER 5

Uranine: Real-time Privacy Leakage Detection without System Modification for Android

5.1. Introduction

Privacy encompasses an individual's or a party's control of information concerning themselves. With the advent of smartphones and tablets, third party applications play an important role in the lives of individual consumers and in enterprise businesses by providing enriched functionality and enhanced user experience but have simultaneously also led to privacy concerns. On the consumers' side, how third-party applications deal with the wealth of private data stored on the mobile devices is not quite clear. Enterprises, on the other hand, need to protect sensitive business data. With the implementation of Bring Your Own Device (BYOD) policies to better accommodate the needs of employees, the issue is further aggravated as the business data is stored on devices that are not completely trusted. Leakage of business data to the Internet or from business applications to personal applications is an important concern. Some leakage of private data may be legitimate and even intended; yet, other leakages may be questionable. We therefore believe that information about the privacy leaks should be completely transparent and accessible to the user (or the IT administrator in case of enterprises). The user may then choose to allow or disallow the leaks either through real-time interaction with an on-device controller or through preset policies. In particular, apart from good accuracy and performance, the detection of privacy leaks should have the following requirements.

- *Real-time*. Real-time detection, or detection right at the time leaks happen, enables situationally-aware decision making. The situation (condition) under which a leak happens is important; a privacy leak may be user-intended and in that case legitimate. For example, upload of a user's address book to a social network under user's consent is legitimate. Offline detection of leaks may be helpful but does not usually identify the complete situation under which a leak happens.
- *No system modification*. Mobile devices typically come locked and it is beyond an average user to root or unlock the devices to install a custom firmware.
- *Easily configurable*. The user should be able to enable the privacy leak detection just for the apps she is concerned about. Other parts of the device such as system server processes and possibly some trusted apps from the device vendor should run without overhead.
- *Portability*. The framework should work across different devices with potentially different architectures, e.g., ARM and x86, and with different runtimes, e.g., Dalvik and ART (a recently introduced Android runtime[1]), with little or no code modification.

There have been many earlier systems targeted at detecting privacy leaks but all have some drawbacks with regards to the above characteristics. TaintDroid [48] detects privacy leaks in real-time but requires the installation of a custom Android firmware, which is possibly only for the most expert users. Furthermore, TaintDroid's firmware code modifications must be adapted to different architectures and even different Android versions. Phosphor [24] is a dynamic taint tracking system for Java and can work on Android. It instruments the application and library code to detect privacy leaks in real-time. However, it requires modification of bytecode of platform libraries, which again requires custom firmware and hinders wide-scale deployment. Static analysis systems fail on the real-time requirement; inputs from the user or from the remote server may affect what is sent out

[1]https://source.android.com/devices/tech/dalvik/art.html

Table 5.1. A comparison of Uranine with dynamic approaches. + is better, − is worse.

	TaintDroid [48]	Phosphor [24]	Uranine
Real Time	Yes (+)	Yes (+)	Yes (+)
System Modification	Yes (−)	Yes (−)	No (+)
Configurability	Little (−)	Little (−)	High (+)
Accuracy	Good (+)	Good(+)	Good (+)
Performance (runtime)	Good (+)	Good(+)	Good (+)
Portable	No (−)	Yes(+)	Yes (+)

of the device and thus the leak may or may not be considered legitimate. Offline dynamic analysis systems such as AppsPlayground [120] have false negatives due to incomplete code coverage and by definition are not real-time.

In this chapter, we propose *Uranine*, a real-time system for monitoring privacy leaks in Android applications without platform modification. Our detection of privacy leaks involves soundly tracking information flow at runtime. The major challenge comes from the requirement of no platform modification, including not instrumenting framework code:[2] we need to approximate flow through the framework code and for callbacks, i.e., the application code called by the framework code. This is further complicated by the existence of heap objects, which often point to other heap objects and whose effects may easily lead to missing leaks if care is not taken.

Uranine provides a framework for instrumenting stock Android applications without the need of application source code. It begins by converting the application bytecode to an intermediate representation, which it instruments employing the techniques presented in this chapter. The instrumented IR is assembled back to a new application installable on an Android device. As the instrumented application runs, privacy leakages are automatically tracked.

Apart from the benefits described above, our approach also brings the added benefit of instrumenting just the apps that the user is concerned about; the rest of the system, including the

[2]Throughout the chapter, app code refers to the code contained in the app; framework code refers to the code defined in the Android platform and may be called through Android APIs.

middleware and other apps, run without overhead. Finally, since we do not touch the Android middleware and the Dalvik runtime, our approach ensures portability. Table 5.1 summarizes the comparison between Uranine and other similar systems.

We note that Uranine is a framework for information-flow tracking. While we present it here for privacy leakage detection, it may be used for other applications such as hardening applications against information flow vulnerabilities.

This work makes the following contributions.

- We solve the problem of tracking private information through platform APIs and libraries without modifying the platform by developing appropriate data structures and algorithms in Section 5.3.1.1.
- The Java language and especially the Android platform builds around callbacks, functions in app code that are called by the platform libraries. We discuss the challenge of handling callbacks for real-time information flow tracking and propose the first solution for this problem in Section 5.3.1.2.
- Aspects of Java, including reference semantics for objects and garbage collection, pose a problem with regards to developing a clean solution that does not interfere with the Java model. Our solution centered on hashtables with weak references to hold necessary data-structures for different objects solves this problem (Section 5.3.1.3).
- We have developed a system prototype for Uranine for real-time detection of privacy leakages in Android apps without system modification. The implementation involved addressing multiple challenges (Section 5.4 and is likely the most sophisticated bytecode manipulation framework for Android to date.

We evaluated (Section 5.5) a working prototype of Uranine on both DroidBench, a benchmark suite developed by researchers, and real-world applications from Google Play. The evaluation shows that Uranine is accurate in tracking information flows. Our evaluation of performance overhead shows that Uranine has acceptable overhead on real-world applications. We also note that it is possible to further reduce the performance overhead of Uranine by performing static analysis and instrumenting only paths along which private information flows can take place.

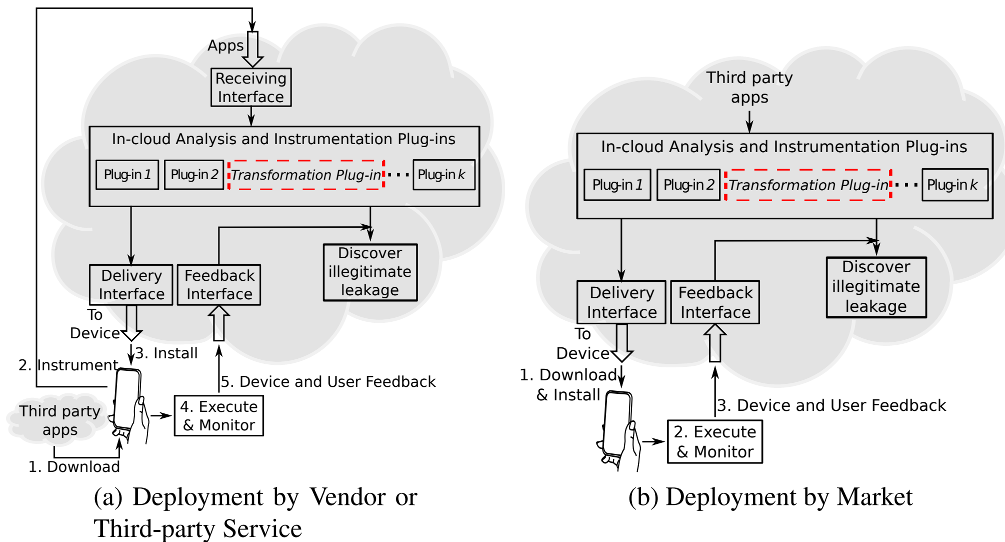
The rest of this chapter is organized as follows. Section 5.2 gives the background and states our approach together with the challenges involved. A detailed description of the Uranine framework is covered in Section 5.3 while Section 5.4 covers the implementation aspects. Section 5.5 gives our evaluation of Uranine. We then have some relevant discussion in Section 5.6 and related work in Section 5.7. We finally conclude in Section 5.8.

5.2. Background and Problem Statement

This section gives a brief background of Android and privacy leakage detection and then goes to identify the problem, how a solution could be deployed, and the associated challenges.

5.2.1. Android Background

The Android OS is based on the Linux kernel and implements middleware for telephony, application management, window management, and so on. Applications are typically written in Java and compiled to Dalvik bytecode, which can run on Android. The bytecode and virtual machine mostly comply with the Java Virtual Machine Specification. Unlike the JVM, The Dalvik Virtual Machine is a register-based VM. Each method has its own set of registers (not overlapping with other methods). Instructions address these registers to perform operations on them.



(a) Deployment by Vendor or Third-party Service

(b) Deployment by Market

Figure 5.1. Deployment by Vendor or Third-party Service

5.2.2. Problem Statement

Static analysis has its own advantages for information flow tracking but a dynamic information flow tracking solution may also be desirable for the following reasons: (a) static analysis may only tell what may happen but cannot tell what actually happens. Runtime conditions, including inputs from the user and the server may influence what actually happens and so that any privacy leaks may be classified as legitimate or illegitimate. Even if a static analysis can detect user interaction, what exactly a user confirms is very difficult for it to capture. (b) Private sources in Android, which are based on URIs, such as contacts, cannot be soundly tracked by static analysis (unless it marks all database queries as possible private sources). Databases such as contacts are accessed through corresponding URIs, which are but wrapped strings and may be obfuscated or inaccessible statically. Lastly, (c) Static analysis is often conservative due to scalability reasons and may have false positives. In the light of all these points, we focus on dynamic information flow tracking.

Previous dynamic analysis approaches on Android for tracking information flow have modified the Dalvik VM or the library code to propagate taints [24, 48]. As this requires platform modification and thus limits the usability, we question if dynamic information flow tracking is possible without platform modification by rewriting the apps alone. Uranine answers this question positively. It accepts stock apps from the user, and returns a ready-to-run instrumented app enabled with information flow tracking.

5.2.2.1. Deployment Models. Figure 5.1 shows the two possible deployment models of our approach. The first model is suitable when there is no control on the source of apps. It is suitable for enterprise, third-party subscription services, individual users, and smartphone vendors and carriers. As the user downloads a third-party app, the downloaded app is passed to our system for instrumentation. Such a system would typically reside in the cloud as a service supported by the vendor or a third-party. It is also possible to place this service on the users' own personal computers or enterprise's servers. Once the app has been analyzed and instrumented by the system, it is installed on the user's device. The app is then constantly monitored on-device as it runs. We note that the whole process may be completely automated with the use of an on-device app so the user needs to only confirm the removal of the original app and installation of the instrumented app. Furthermore, laymen users may be provided with preset information flow tracking and enforcement policies.

The second deployment model is suitable for application markets or enterprise application stores, which want to instrument all apps with some general security policies to protect the consumers. In this case, the apps are instrumented in the cloud before the user downloads and installs the app. Thus, there is no need for the original app to be uploaded from the device to the service for instrumentation. There is no other difference between the two models. We note that other existing real-time taint-tracking systems do not have similar deployment models.

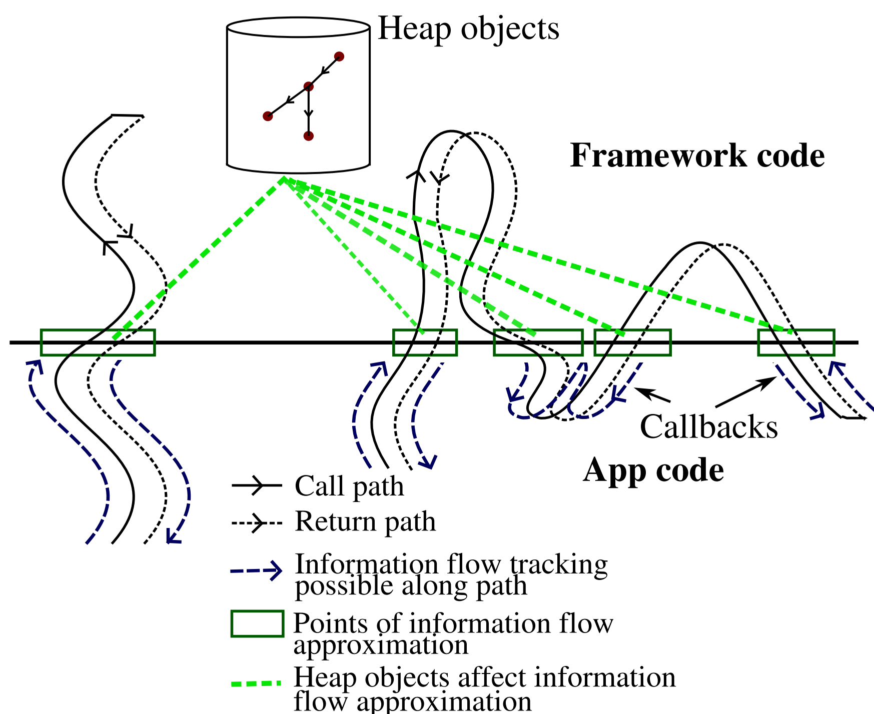


Figure 5.2. A depiction of challenges C1 and C2 met in Uranine. There are paths between app code and framework code depicted as meandering function call paths and return paths, together with callbacks (the app code that is called by framework code). The left path results from ordinary calls while the right path includes callbacks. Information flow tracking can only be done for app code, requiring approximations for framework code. Callbacks must be handled soundly. Objects on the heap point to each other and their effect on information flow should be properly accounted for during approximations.

Android apps are digitally signed by their developers and so instrumenting an app would require an application to be re-signed. The current app update system at Google Play (and possibly other Android markets) depends on apps' signatures. Deployment by third party services will therefore have to provide out-of-band mechanisms to notify users of available updates. This is however not much of a concern: mobile app management and app wrapping products such as Good [63] and MobileIron [100] already provide similar deployment models to enterprises in the context of API interposition similar to [42, 43, 140].

5.2.2.2. Challenges. Following are the challenges that we solve in Uranine.

C1 Framework code cannot be modified. This means we cannot instrument framework code. We summarize the effect of framework APIs according to a custom policy, combined with manual summarization for a few special cases. Previous works on static or dynamic binary instrumentation [105, 141, 151] have needed to summarize system calls or very simple functions in low-level libraries like libc, which are much simpler. Static analysis works also typically use summarization [60, 93] to achieve scalability. However, we show by example that in our context of dynamic analysis and complex framework with Java data structures in Android, summarization alone is not sufficient. Heap objects can be particularly challenging to handle and we need additional techniques for effective taint propagation.

C2 The effect of callbacks should be accounted for. Callbacks are functions in app code that may be invoked by the framework code. Since framework code cannot be instrumented, we cannot do taint propagation when callbacks are invoked. We propose a technique, using over-tainting to avoid false negatives. Challenges C1 and C2 are represented in Figure 5.2.

C3 In the Java language model, objects follow reference semantics and so we must have a way to taint the locations referenced. Furthermore, objects are deallocated automatically by garbage collection; so our taint-tracking data structures should not interfere with garbage collection.

As noted above, there are trade-offs between system modification and detection accuracy. However, we note that even though we resort to over-tainting to solve some of the above challenges, our results demonstrate that a carefully conceived design may still have a low false positive rate in practice. We discuss our solutions in detail in the next section.

Figure 5.3. Instrumentation flow in Uranine

5.3. Uranine Design

Uranine offers a general framework for instrumenting applications statically and for providing information flow tracking, which may be used in a number of applications, including tracking privacy leaks and hardening applications against vulnerabilities. Figure 5.3 depicts the architecture of Uranine. When an app is given to Uranine, the app code is first converted to a custom intermediate representation (IR) that can be instrumented for taint propagation to happen at run time. The instrumented IR is then converted back to bytecode and a new app is prepared. Since the framework code cannot be instrumented, it approximates the effects of framework code through a few general but customizable summarization rules. The rest of this section first describes our techniques for taint storage and propagation and the instrumentation details. The latter part of the section then describes our static analysis.

5.3.1. Taint Storage and Propagation

The techniques of taint storage and propagation influence the accuracy and runtime performance of privacy leakage detection. Our techniques focus on providing privacy leakage detection without false negatives under the constraints of not modifying the platform. Much of the design for taint tracking here is fairly routine and may be found in previous work [24, 48, 127]. We describe the routine or obvious aspects very briefly and then discuss in detail the specific challenges and corresponding solutions in our work.

Each entity that may be tainted is associated with a taint tag, which identifies what kind of private information may be carried by the entity. In the Uranine model, taints are stored and propagated for local variables (i.e., method registers), fields, method parameters and returns, and objects. Different bytecode instructions handle different storage types (i.e., local variables, fields and so on) and accordingly have different taint propagation rules. Additionally, in a complete system, IPC (inter-process communication) taints and file taints may be handled at a coarser granularity. For IPC, the entire message carries the same taint. Similarly, an entire file is assigned a single taint tag. In our design, tracking IPC and file taints requires communication with an on-phone Uranine app, which keeps track of all file taints and IPC taints from instrumented applications. We focus on taint tracking withing Java code (more specifically, Dalvik bytecode) and further discussion on IPC and file taints is out of scope of this work.

We next describe the taint propagation rules for the different situations. We begin our discussion by assuming we can instrument all the code (including the framework) and then introduce changes that would be required to leave the framework code intact.

Method-local registers. For each register that may possibly be tainted, we introduce a shadow register that stores the taint for this register. Any move operations simply also move the shadow registers. The same also happens for unary operations while for binary operations, we combine the taints of the operands and assign to the shadow register of the result. Instructions assigning constants or new object instances zero the taint of the registers.

Heap objects. Heap objects include class objects, containing fields, and arrays. For each field that may possibly be tainted, we insert an additional shadow taint field in the corresponding class. The load and store instructions for instance fields and static fields are instrumented to assign to or load from these taint fields to the local registers. We note that we may not insert additional fields

into framework classes. In this case we taint the entire object. How this is done and the effects of this will be discussed shortly.

In the case of arrays, each array is associated with only a single taint tag. If anything tainted is inserted into an array, the entire array becomes tainted. This policy is used for efficiency reasons and has been also adopted by other works such as TaintDroid. We also support index-based tainting so that if there is an array-get (i.e., a load operation) with a tainted index, the retrieved value is tainted. We will discuss shortly how we associate taint with Array objects.

Method parameters and returns. Methods may be called with tainted parameters. In this case, we need to pass on the tainted information from the caller to the callee. We take a straightforward approach to achieve this; for each method parameter that may be tainted, we add an additional shadow parameter that carries the taint of the parameter. These shadow parameters may then convey the tainted information to the local registers. Method returns are trickier. Since we can return only one value, we instead introduce an additional parameter to carry the taint of the return value. In Java, we have call-by-value semantics only so that making assignments to the parameter inside the callee will not be visible to the caller. We therefore pass an object as the parameter, which is intended to wrap the return taint. The caller can then use this object to see the return taint set by the callee.

Our next part of discussion relates to specific challenges discussed in Section 5.2 and mostly relates to requirements of not changing the framework code.

5.3.1.1. Calls into the framework (Challenge C1). Whereas the application code may be instrumented for taint propagation, we may only approximate the effects of calls into the framework code on taint propagation. We use a worst case taint policy to propagate taints in this case:

- Static methods. For static methods with void return, we combine the taints of all the parameters and assign this to all the parameter taints. For static methods with non-void

returns, the taints of all the parameters are combined and assigned to the taint of the register holding the return value.

- Non-static methods. Non-static methods often modify the receiver object (the object on which the method is invoked) in some way. Therefore, we combine the taints of all the non-receiver parameters; apart from its original taint, the receiver object is now additionally tainted with this combined taint. In case the method returns a value, the return taint is defined as the receiver taint.

Note that these rules are not enough to summarize the effects of framework code. Non-static methods often have arguments that are stored into some field of the receiver. Consider the following piece of code.

```
List list = new ArrayList();
StringBuffer sb = new StringBuffer();
list.add(sb);
sb.append(taintedString);
String ret = list.toString();
```

In this case, `sb` and `list` are untainted until line 4. Thereafter, `sb` is tainted and `ret` should be tainted because it will include the contents of `taintedString`. Our general solution is that when an object becomes tainted, any objects containing that should also become tainted. For every object o_1 that may be contained in another object o_2, we maintain a set of the containing objects. If the taint of o_1 ever changes, we propagate this taint to all the containing objects. The set of containing objects is updated whenever we have a framework method call $o_2.meth(.., o_1, ..)$, where $meth$ is a method on o_2 and possibly belongs to the framework code. This is a worst case solution; in certain cases, such a method would not lead o_1 to be contained in o_2. The update operation may be recursive, so that an update to taint of o_2 may lead to updating the taint of the

objects containing o_2, and so on. Objects may point to (contain) each other and hence there may be cycles; the update operation will however achieve a fixed point at some point and then terminate.

5.3.1.2. Handling callbacks (Challenge C2). A callback is a piece of code that is passed onto another code to be executed later on. In Java, these are represented as methods of objects that are passed as arguments to some code, and the code may later invoke methods on that object. These objects typically implement an interface (or extend a known class) so that code is aware which methods are available on the object.

Android makes an extensive use of callbacks, which often serve as entry points to the application. Examples of such callbacks are `Activity.onCreate()` and `View.onClick()` when overridden by subclasses. Apart from these, callbacks may be found at other places as well. For example, `toString()` and `equals()` methods on objects are callbacks. Identifying callback methods correctly may be done using class hierarchy analysis. A class hierarchy analysis analyzes the inheritance relationships between different classes and, based on these results, the overriding relationships between different methods. The class hierarchy analysis acts as a guide to the rest of the instrumentation by defining how different methods are dealt with during instrumentation.

Since callback methods override methods in the framework code, their method signatures may not be changed to accommodate shadow taint parameters and returns, lest the overriding relationships are disturbed. For example, consider the following class.

```
class DeviceIdKeeper {
  private String id;
  public DeviceIdKeeper(TelephonyManager m) {
    id = m.getDeviceId();
  }
  public toString() { return id; }
}
```

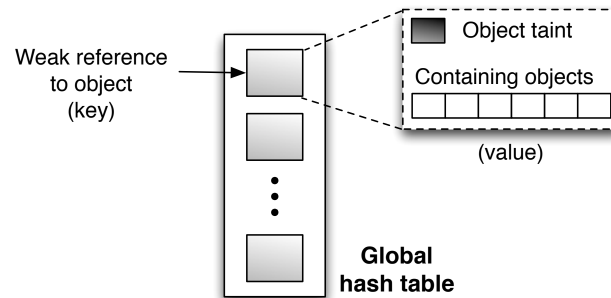


Figure 5.4. Associating taint data-structures with objects

The app code may call `toString()` on a `DeviceIdKeeper` instance. Since the return here may not be instrumented to propagate taint, we may lose the taint here. Furthermore, it is also possible that this method is called at some point by the framework code.

Our solution. In order to not lose taint in this case, our solution is to lift the return taints of all callback methods to the receiver objects. That is, in the instrumented callback method, the return taint is propagated to the receiver object taint. In case a possible callback method is called by app code with tainted parameters, we taint the receiver object with the taint of the parameters and then inside the method definition taint the parameter registers with the taint of the receiver. Since heap objects can carry taints in our model, such over-tainting needs to be done only in case of parameters of primitive types. With the parameter and return tainting in place, we may use the techniques described for calls into the framework (Section5.3.1.1) to summarize the effect of this call. The key to note here is that the receiver object of the callback serves as a handy taint carrier and thus taint is not lost in both the cases: when the callback is called by an app method, and when it is called by the framework.

5.3.1.3. Taint data-structures (Challenge C3). From the above, it is quite clear that we need a way to taint objects. Java uses reference semantics to address objects. That is, object variables are pointers to object values on the heap and assignment for objects is only a pointer copy. Thus, we

may have two types of tainting, either tainting the pointer, or tainting the object. Storing pointer taints is simple and has been discussed as storing taints for method-locals and fields. In addition, we also need to associate a set of containing objects with each object (Section 5.3.1.1).

Our solution. In our solution, we use a global hashtable, in which the keys are objects and the values are records containing their taints and the set of containing objects. Any time, the taints or containing objects needs to be accessed or updated, we access these records through the hashtable. Our hashtable uses weak references for keys to prevent interference with garbage collection. In Java, heap memory management is automatic; so we cannot know when an object gets garbage-collected. Weak references are references that do not prevent collection of objects and so are ideally suited for our applications. We further note that these data-structures should allow concurrent access as the instrumented app may have multiple threads running simultaneously. A schematic of our global hashtable is presented in Figure 5.4.

We considered but rejected an alternative method of keeping these data structures. With every object, we can possibly keep a shadow record, which is an object that stores the object taint and the set of containing objects in its fields. The instrumentation may then move this shadow record together with the main object through method-local moves, function calls and returns, and heap loads and stores. This technique however does not work well with the way we handle calls into the framework. Consider the following code fragment.

```
// list is a List
// obj is an object
list.add(obj);
obj2 = list.get(0);
```

In the above code, `obj` and `obj2` could be the same objects. However, since the loads/stores and moves inside the `List` methods are not visible to us, we cannot track the shadow record of

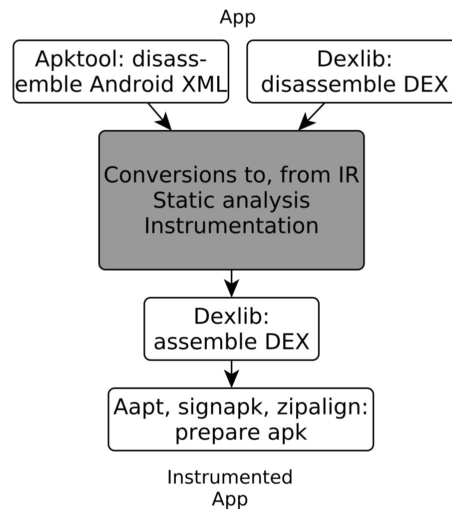


Figure 5.5. Uranine implementation depicting the use of existing code (white boxes) and the features we implemented (gray, discussed in detail in Section 5.3).

`obj` there. The shadow record of `obj2` may at most depend on the record of `list`. Thus, there is no way to make the shadow records of `obj` and `obj2` the same, something that we achieve easily with our approach of weak hashtables.

5.4. Implementation

5.4.1. Toolchain and IR

We have implemented a working prototype of Uranine. We use a library called dexlib [12] to disassemble and assemble Dalvik bytecode. The disassembled representation is converted to an intermediate representation (IR). In addition, we also use apktool [1] to disassemble the binary Android XML format (needed for discovering entry points for static analysis) and other tools from the Android SDK and elsewhere to prepare an instrumented app. Figure 5.5 provides these details graphically.

We choose to work on an IR very close to the bytecode and do not require decompilation to either Java bytecode or the source code as some previous works have required. Since decompilation is not always successful, this approach improves the robustness of our system. Disregarding details like register widths, the Dalvik bytecode instructions[3] generally have a direct correspondence with the instructions in the IR. Similar instructions (such as all binary operations or all kinds of field accesses) are represented as variants of the same IR instruction. Range instructions (`invoke-*/range` and `filled-new-array-*/range`) access a variable number of registers; these are converted to the simple representations of `invoke-*` and `filled-new-array-*` instructions with a variable number of register arguments in the IR. Even though we use this IR for instrumentation, it is also suitable for performing static control flow and data flow analysis. In fact, the same IR is used as input to our class hierarchy analysis, the results of which then guide the instrumentation. The instrumented IR is then finally assembled back to Dalvik bytecode.

Most of our instrumentation code is written in Scala, with about a hundred lines of Python code. The taint-tracking data structures and related code is written in Java. The instrumentation adds a compiled version of this code to every app for runtime execution. The total Uranine codebase sizes to over 6,000 lines of code. We note that Scala allows for writing terse code; the equivalent Java or C++ code is usually two to three times as long.

5.4.2. Instrumentation Details

The instrumentation of instructions for information-flow tracking at the Dalvik byte-code level with our design is particularly challenging. Some of the challenges we handled during implementation are:

[3]http://s.android.com/devices/tech/dalvik/dalvik-bytecode.html

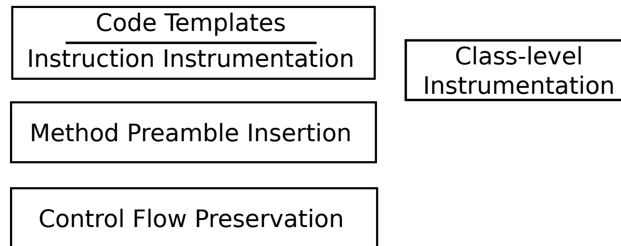


Figure 5.6. Instrumentation organization

- Handling exceptional control flow. Many instructions such as method invokes and array and field accesses can lead to exceptions. The Dalvik verifier pessimistically assumes that such exceptional control flow will be there in the presence of such instructions. This leads to failures when it type checks registers (the verifier implements a primitive type checking where it checks integer, object, and wide register types). We must therefore exercise additional care so that the added instrumentation instructions do not alter the control flow possibilities.
- Limited register addressing. Several frequent instruction opcodes permit only four-bit registers, implying that only the first sixteen registers can be addressed in those instructions. Thus, one has to prudently write the instrumentation instructions, sometimes even shifting registers around to get data into an addressable register.
- Limited context. This challenge comes because we choose to simplify the implementation by not knowing the current type of every register. This means that for instrumenting a given instruction, we can use only the registers specified in that instruction and a few temporaries.
- Temporary allocation. A few temporary registers are needed when moving registers, for invoking object-tainting functionality, and so on. We keep a count of the maximum temporary registers needed and provide necessary instrumentation at the beginning of the methods.

The above challenges make our instrumentation system more complex than any other systems on Android that we know of [?, 42, 67]. We organize our implementation in three-layered stack as shown in Figure 5.6, where each layer addresses some of the challenges. The top layer deals with individual instruction instrumentation. Even though each instruction opcode has a specific way of instrumentation, certain sequences are common and we extract them out as templates, which promote code modularity. For instance, consider the following Scala snippet for instrumenting the field put instruction.

```
val (s0, s1, setTaintMoves) = templates.generateMoves(r2, taintRegOf(r1), aType1 =
    NAType)
lst ++= setTaintMoves
lst += templates.setTaint(s0, s1)
```

The prime purpose of this code is to update the taint of object referenced by register `r2` with the taint of register `r1`. This requires calling a function. The method invoke instruction has certain restrictions (if parameter registers are addressed with more than four bits they must be consecutive), and so register moves into temporaries may be required. The templates generate the appropriate instrumentation instructions while helping keep code organized and simple.

The second layer inserts a method preamble (some instrumentation code at the beginning of a method) that takes care of allocating the requisite number of temporary registers. In Dalvik bytecode, method parameters are stored in the last registers. Thus, if the register count of a method is increased, the parameters need to be explicitly shifted to their original position so that original instructions are not disturbed. After this shift, the last registers are available to be used as temporary registers.

The third layer takes care of the first challenge, ensuring that the original control flow in the method is not disturbed. It does this by inserting as many `catchall` clauses around the newly

Table 5.2. Accuracy evaluation of Uranine and comparison with TaintDroid

App	Uranine	TaintDroid	App	Uranine	TaintDroid
mobi.android-cloud.app.ptt.client	Contact	Contact	com.ama.lovetest.cal-culator	IMEI, Phone#	IMEI
com.enlightened.An-droidskyjewelsfree	IMEI	None	com.flashlight.tre-film.coins	IMEI	IMEI
com.magmamobile.ga-me.Slots	IMEI	None	com.silkenmermaid.g-au.dldic	IMEI	IMEI
me.zed_0xff.android.al-chemy	IMEI	None	com.gamevolution.Ma-rbleMadnessPro	IMEI	IMEI
com.magmamobile.ga-me.BubbleBlast2	IMEI	None	com.reverie.game.toilet-paper	IMEI	IMEI
com.rhs.wordhero	Loc	Loc	com.red.white.blue.free	IMEI	IMEI
com.rferl.almalence.st-aringcat	IMEI	IMEI	com.gameloft.andro-id.ANMP.GloftGTFM	IMEI	IMEI
app.win.conforl1	ICCID, IMEI, Phone#	None	com.alloright.trib	IMEI, Loc, Phone#	IMEI, Loc
com.anbgames.open-thedoor.hoola2	IMEI	IMEI	com.euro2012.geekbea-ch.acquariusoft	IMEI	IMEI
com.aceviral.top-truckfree	IMEI	IMEI	com.fjj24512014.korea	IMEI	None
com.flirtalike.android	IMEI, ICCID	IMEI, ICCID	net.aaronsoft.poker.eva	IMEI	IMEI
com.keithe.lwp.aq-uarium	IMEI	IMEI	com.mobizi.scratchers	IMEI	None
com.androiminigsm.fs-cifree	Contact, IMEI	Contact, IMEI	sg.vinay.FourpicsOne-wordcheatsanswers	IMEI, Phone#	None
mobi.jackd.android	Loc	Loc	com.electricpocket.rin-go	Contact	Contact
com.topface.topface	IMEI	IMEI	com.keek	IMEI, ICCID	IMEI
com.pilotfishme-diainc.happyfish	IMEI, Loc, Phone#	Loc	com.phantomefx.re-eldeal	IMEI	IMEI

added instrumentation instructions (we are sure that the new instructions will not misbehave) as required so that the verifier can type-check correctly.

In addition to this stack, we also have class-level instrumentation, which inserts fields, changes method names, and so on.

5.5. Evaluation

We evaluate Uranine on two aspects: accuracy and performance overhead. To perform accuracy evaluation we configured Uranine to detect the leakage of location, phone identifiers (like

IMEI and phone number), and contacts (address book). We can similarly add other private information sources as well. Our sinks include all APIs that send data to the network, write to the file system, or send SMS messages.

Our app dataset consists of 1,490 apps randomly selected from Google Play. Apps are instrumented automatically and run with random inputs (fuzz testing) provided by Android Monkey tool. For understanding privacy leakage results, we also conducted manual tests for a smaller set of apps.

5.5.1. Accuracy

In the accuracy evaluation, we measure whether Uranine is able to detect any privacy leaks and how does it fare on the false positive rate. We performed two distinct experiments: one using the DroidBench benchmarking suite [19] and the other using real-world applications collected from Google Play and comparing the results with TaintDroid.

5.5.1.1. DroidBench. DroidBench is an open-source suite of simple Android applications to test information flow analysis tools on Android. Many applications contain data leaks similar to what might be seen in the real works, while others present scenarios meant to confuse analysis tools into flagging false positives. Even though originally meant for testing static analysis tools, the suite can serve as a ground for testing dynamic analysis tools such as Uranine as well.

Out of a total of 61 applications in DroidBench 1.1, we tested all test-cases except 15, which relate to implicit flows and Android lifecycle. Implicit flows are a general limitation for dynamic analysis whereas many Android lifecycle leaks happen through persistence of state in files, leading us to miss such leaks as we do not handle taint propagation through files and other persistence mechanisms in our prototype. We also did not test 3 Android-specific test cases related to leaking of

passwords (we do not have a systematic way of identifying password fields) and taint propagation through files.

In testing with Uranine, most test cases were either true positives or true negatives. We discuss here those where the detection was either a false positive or a false negative. The four ArraysAndLists test cases tested placing both untainted and tainted data in data structures and then accessing untainted data from them and leaking it. Because of our decision to taint at the granularity of arrays and framework objects (Lists are framework objects), the results were expected to be false postives, and so was the outcome. We also encountered a false negative in the Generaljava/Exceptions4 test case, which constructs an exception with a tainted parameter and throws it. The exception handler subsequently leaks the tainted parameter. The false negative was due to a wrong assumption that we made in the implementation: we assumed that a newly constructed object has only one possible reference (the local register to which they are assigned), that all subsequent references will be derived from this reference, and thus tainting this reference (instead of the object) is sufficient. The reasoning is flawed because exceptions will get a new reference not connected to the original reference. This bug was easily fixable. Finally, there was a false negative in a reflection test-case because the instrumentation could not properly determine taint propagation through a method of reflective class (the other three reflection test cases passed).

To summarize, Uranine met our expectations in our evaluation of DroidBench.

5.5.1.2. Google Play applications. In this section we evaluate how Uranine performs in detecting privacy leaks in real-world applications. An experiment on real-world applications is important because a benchmarking suite may miss some kinds of leakage scenarios and also because we need to understand the false positive and false negative rates in the wild.

Unlike DroidBench, Google Play applications do not come with ground truth of what they would leak. We therefore used TaintDroid results to compare with our results. Our methodology

involves running Uranine-instrumented applications on a TaintDroid build. This allows us to generate both TaintDroid's and Uranine's results together in one run, thus eliminating any differences that may come because of random inputs or non-determinism in multiple runs.

Manual tests. We conducted manual tests on a physical device (Samsung Nexus S) over a small, random subset of apps. These results enable us to carefully study the differences between TaintDroid and Uranine. The results are depicted in Table 5.2. The results, where neither TaintDroid nor Uranine detected any leakage, are not shown in the table.

Our results show some disagreement with TaintDroid. We see that TaintDroid does not detect any phone number leaks that we detect; a look into TaintDroid code then revealed to us that TaintDroid has disabled tracking of phone numbers with the comment "causes overflow in logcat, disable for now" in source code. In all other cases of disagreement between Uranine and TaintDroid, we manually confirmed the correctness of Uranine. It turns out that in the cases where Uranine does detects an IMEI (or ICCID) leak while TaintDroid does not, there is some kind of hashing of the identifier involved, such as the calculation of MD5 or SHA1 digests. It appears that TaintDroid does not propagate taint across the functions that calculate these digests. This is also confirmed in AppsPlayground [120]. In conclusion, our results are generally consistent with TaintDroid. Any apparent inconsistencies result from implementation artifacts of taint tracking. It is worth emphasizing here that our contribution is not to show an improvement over other systems in terms of detecting more privacy leaks but to do the detection without system modification.

Automatic tests. We further conducted automatic, random testing on a bigger dataset of 1,490 apps. The tests were conducted on Android emulator (provided with the Android SDK) running a TaintDroid image. Since the emulator does not provide most of the device identifiers (such as IMEI and phone number), we further added some code to our emulator image to provide real-looking

Table 5.3. Leaks detected in automatic tests

Leak type	Apps leaking	Leak type	Apps leaking
IMEI	310	IMSI	18
ICCID	16	Phone #	79
Location	107	Contacts	5

identifiers on the respective APIs for accessing these identifiers. Because of these modifications, our emulator's TaintDroid can also detect phone number and IMSI leaks.

Our runs detected privacy leaks in a total of 360 apps; in the rest of the apps, no leak was detected either by TaintDroid or Uranine. The results for TaintDroid and Uranine differed for 177 apps. We have manually analyzed each of these cases, and have found that Uranine was accurate in most cases. Below, we detail our findings and bring out relevant insights.

For 92 apps where Uranine detected privacy leaks but TaintDroid did not, we confirmed that these were TaintDroid's false negatives. In all these cases, the apps leak the device identifiers after hashing (with, for example, MD5). In most cases, we were able to see the MD5 checksum of the device identifier being leaked (IMEI leaks were most frequent) in plaintext. Further, in other cases, these leaks were in ad libraries that are known to have the leaks flagged by Uranine. For example, our analysis of an older version of Admob library shows that it leaks the MD5 of a string derived from the phone's IMEI number.

Uranine's detection of leakage in 4 apps is likely to be a false positive. In two apps, our logs reveal Uranine flags leakage when an empty string is being written to a file. In the other two cases, Uranine detects IMEI leakages on writing strings that look like base64 codes. Decoding those codes however does not reveal the IMEI number nor anything that looks like a hash of that. False positives are actually expected in Uranine, due to overtainting as part of our design. Considering this, 0.2% false positives are insignificant.

There was another set of 13 apps where Uranine flagged leakage but TaintDroid did not. In all these cases, we can see strings looking like MD5 or SHA hashes being leaked, but were unable to derive them from known identifiers (perhaps they were mixed with some salt before hashing). Though we could not classify these cases, we believe them to be TaintDroid false negatives. Finally, we detected 14 cases that were false negatives for Uranine; we could however correct them by adding additional sinks that we missed earlier.

In summary, we found Uranine to be fairly accurate in detecting privacy leaks with few errors. Table 5.3 shows the privacy leaks detected by Uranine.

5.5.2. Performance

Measuring the runtime overhead of applications instrumented by Uranine is not trivial. First, there are no popular macrobenchmarks for Android. The DaCapo benchmarks [27], which are popular Java benchmarks, are not easily ported to Android (due to their use of Java-specific libraries and GUI) and moreover, may also differ from real-world application workloads on Android. Second, conventional microbenchmark suites for evaluating virtual machine performance may also give skewed results as we are instrumenting applications here rather than the virtual machine. Nevertheless, for documentation, we present our evaluation on CaffeineMark 3 microbenchmark. For macrobenchmarking, we chose to evaluate certain events from real-world Android applications. All performance evaluation tests were done on a Samsung Galaxy Nexus running Android 4.3.

5.5.2.1. Microbenchmarks. We used CaffeineMark 3 microbenchmark to measure the overhead of our instrumentation. For this purpose, we used com.android.cm3 application from Google Play, that packages this benchmark suite. We ran both the original application and the instrumented version of this application on the device. Table 5.4 presents the results.

Table 5.4. CaffeineMark 3 performance. This benchmark may not adequately represent the real-world performance of Uranine

Benchmark	Original	Instrumented
Sieve	2615	36
Loop	16572	22
Logic	10268	450
String	6373	418
Float	6833	22
Method	5349	216
Overall	6853	94

The above table shows an overhead of about 70 times. Given the complex instrumentation we perform (such as for method calls and fields), we had expected the performance overhead to be high. However, we suspect this benchmark itself also exacerbates the reported overhead. One possible reason is that all the test cases use method calls and have instructions such as move at the bytecode level. All these additional instructions are instrumented, sometimes with high overhead, resulting in a skewed report. For example, the Logic test case contains simple condition checks and branches, which are themselves not instrumented in Uranine; however, the other instruction such as method calls and moves are still instrumented.

It should be noted that the above benchmark does not mix in application code with framework code. In real applications, where a large chunk of framework code, mixes with the application code, we expect the performance to be much better, as is also shown in the following macrobenchmarks.

5.5.2.2. Macrobenchmarks. As discussed earlier, traditional macrobenchmarks are not easy to port on Android. Using real-world Android applications is the next obvious choice. However, most applications are GUI intensive and interactive in nature. Thus, one cannot simply run the benchmark application and obtain the results. We devise our own methodology of evaluating performance of Android applications in response to certain events. For our benchmarks, we select a total of six events from three very popular applications: BBC News, Last.fm (a music application

Table 5.5. Macrobenchmark performance. The reported times (Original/Instrumented columns) are medians over five independent runs.

Benchmark	Event	Original (ms)	Instrumented (ms)	Overhead
BBC News (version 2.5.2 WW)	Launch	953	1418	49%
BBC News (version 2.5.2 WW)	Click (“Live BBC World Service”)	450	434	-
Last.fm (version 1.9.9.2)	Launch	523	567	10%
Last.fm (version 1.9.9.2)	Click (“Sign up”)	132	140	6%
Contacts (from AOSP 4.0.4)	Launch	580	645	11%
Contacts (from AOSP 4.0.4)	Click (“Done” after contact creation)	23	59	156%

with social networking features), and the stock Android application for managing contacts. For each application, we evaluate the time to launch the main activity of the application and the time to complete a click of a pre-selected feature on the application. The time to launch the main activity is as reported by the ActivityManager (part of the Android middleware). The time to complete a click is measured by instrumenting the click handler function to report the interval from its beginning to the point it returns.

Table 5.5 presents the comparison of the original applications and those instrumented for information flow-tracking. As can be seen from the table performance overhead is usually low, almost always within 50% and often around 10%. We attribute this to the fact that the Android framework does most of the heavy-lifting during runtime, from creating the UI to managing the data structures and data stores. Thus, even though we saw a huge performance overhead in the microbenchmark, real-world application overhead appears quite low in comparison. Anecdotally, in our runs, we have seen noticeable performance overheads but the overheads have never been intolerable.

Finally, we would like to reiterate that our approach is highly amenable to static analysis. We expect that in production, a tool such as Uranine will be guided by a static analysis, which will be able to identify that most paths cannot propagate the relevant information flow and thus need not be instrumented.

5.6. Discussion

5.6.1. Static Analysis and Optimizations

We believe that Uranine has great potential for optimizations so that runtime overhead can be minimized. First, it is possible to tune the instrumentation, and perform constant propagation passes to reduce the instrumentation overhead. Second and more importantly, it is possible to perform a static information flow analysis that identifies the paths along which the relevant information flow could take place. Such paths are usually small in number and thus if Uranine instruments those paths only, applications may run with negligible overhead. Note that the use of static analysis does not obviate the need for a dynamic analysis system (Section 5.2.2).

We note that the opportunity for static analysis is present in our approach only, involving no platform modification. Previous work such as Phosphor [24] modify the platform libraries to track information-flow and will therefore not benefit much from optimizing app instrumentation by static analysis.

5.6.2. Limitations

We discuss here our limitations and avenues of future work. While Uranine is good for detecting privacy leaks in legitimate applications, a truly malicious app may be able to evade the system through some of these limitations.

Implicit flows. A fundamental limitation of dynamic taint tracking is the inability to track implicit information flows via control flow [127]. Our work shares this limitation. Static analysis may be used to track control flow. However, this leads to the risk of severe over-tainting. Research is underway to make implicit flow tracking practical [75].

Native code. We currently also do not support taint tracking through native code, which some Android applications include in addition to bytecodes. Previous works such as Phosphor and TaintDroid , and all static analysis works on Android, which only analyze bytecodes, all have this limitation. Taintdroid does not allow third-party apps to load native code, mostly resulting in a crash. Greater research is needed in this area to come up with practical solutions for ensuring safety of apps with native code.

Dynamic aspects of Java. As a limitation of static instrumentation, the dynamic aspects of JVM, such as reflection and dynamic class loading (using `DexClassLoader` or similar features in Android) do not cleanly fit in. These may however be supported in the future in our approach. We may apply worst-case tainting for all method calls made by reflection as we do for other methods. Furthermore, we can instrument calls by reflection and alert the user if they do not pass certain security policies (such as restricting reflective calls to only certain API in the Android platform). Code loaded by dynamic class loading may also not be available during static instrumentation. In a deployment, it may be possible to prompt the user to allow reanalysis whenever dynamic code loading is detected so that an instrumented version of the code being loaded can be created.

Incorrect summarization. Policy-based summarization of framework code, as used in our work not only has the problem of over-tainting but could also result in under-tainting of data passing through APIs that do not fit within those policies. For example, some classes may update a global state when their methods are called. We are not aware of such a situation but such cases could be used to bypass the system. Manual summarization of known cases is obviously one solution. Automatic method summarization is an open research problem in static analysis, any progress there will benefit our cause as well.

5.7. Related Work

Information flow tracking. The closest to our work are TaintDroid [48] and Phosphor [24]. The key advantage of our technique is that we do not require modification of the Android platform as these do.

Dynamic taint analysis has been employed in a variety of applications from vulnerability detection and preventing software attacks [105, 115, 134] and malware analysis [23, 130, 143] to preventing privacy exposures [47, 151]. Schwartz et al. describe dynamic taint analysis together with its applications and the details that may be needed to implement it [127]. Depending on the application, the works cited here employ operating system modification, virtual machine introspection, and so on. We present a general technique for taint tracking in this chapter without modifying the Android platform. Our technique may be used for the above applications, especially when there is a constraint to run applications on an unmodified platform.

Apart from the above works, there are also works doing taint tracking by bytecode instrumentation. Haldar et al. [65] implement taint tracking by instrumenting Java String class. Chandra and Franz [31] instrument the Java bytecode for taint tracking and also provide for tracking implicit flows. Their bytecode instrumentation also uses certain taint tracking data structures. These works share the same limitation as Phosphor as discussed earlier.

There are also a number of related works using static analysis. PiOS [46] uses it to detect privacy leaks on iOS apps. Enck et al. [49] and Gibler et al. [62] decompile Dalvik to Java bytecode and perform static analysis on that using existing tools for Java. FlowDroid [19] also converts Dalvik back to Java bytecode and builds on top of Soot[4] while adding in Android-specific requirements to the analysis. Chex converts Dalvik bytecode to WALA[5] IR and then employs WALA for

[4]http://www.sable.mcgill.ca/soot/
[5]http://wala.sourceforge.net

static analysis [93]. Other similar static analysis efforts [19,60,79] also exist. As discussed earlier, there are disadvantages of static analysis over real-time dynamic analysis.

Static instrumentation. Static instrumentation is not a new technique. It has been used earlier for Android applications [42, 43, 140]. These works have focused on API interposition rather than tracking information flow; the latter is more challenging because of the need to instrument many instructions and to encode the semantics of information-flow tracking. AppSealer [144] statically instruments Android applications to repair component hijacking vulnerabilities. Capper [145] is a follow-up work that detects privacy leakages without platform modification. Both these works are similar to Uranine; however, their taint tracking will have false negatives: they try to address C1 but do not solve it adequately and do not even discuss C2 and C3. Instrumentation has been used in other applications as well, some of which even use static analysis to optimize it. Saxena et al. use static analysis to make their binary instrumentation efficient [126]. Xu et al. [141] instrument C sources for taint tracking and further optimize it using static analysis. Slowinska et al. [131] instrument binaries to prevent buffer overflow attacks. AppInsight is another system that uses static instrumentation [121]. It instruments Windows Phone applications to identify critical paths in them as they are used in the wild. The results can subsequently be used to optimize the apps.

Other related work in mobile device security. Kirin [50] defines security policies based on Android permissions. Permissions however are often coarse-grained and also do not give any indication of what may happen at runtime. A number of works additionally prevent access of private information or supply fake data to apps [26, 103, 150]. Information access however does not imply information leakage. For example, an application may have legitimate need to access user's contacts (say for providing editing functionality) but should not leak them out. Furthermore, dynamic taint tracking can also establish the final recipients of private information, e.g., remote servers and so on. Another line of works [22, 87] investigates the user perceptions as related to

mobile privacy. They conclude that users are often not aware of privacy leakages and that proper awareness and usable controls can mitigate users' concerns about privacy.

5.8. Conclusion

This chapter describes Uranine, a framework for dynamic privacy-leakage detection in Android applications without modifying the Android platform. To achieve this, Uranine statically instruments Android apps only and does not need support for information flow tracking from the platform. We provide a design and implementation of Uranine and have evaluated its performance and accuracy. Our results show that Uranine has good accuracy and incurs acceptable performance overhead.

CHAPTER 6

Conclusion

Entire ecosystems are based around modern operating systems such as Android. Different parties such as OS developers, vendors, application developers, carriers, users, and so on come together and play crucial roles in these ecosystems. With all these parties having conflicting interests, it is difficult for them to agree upon solutions to improve the security and privacy properties of these ecosystems. In this dissertation, I presented four works that show that it is possible to deploy solutions without support from the OS developers, vendors, application developers, etc., and still gain security and privacy benefits. Without the support of operating system and the applications, it may at the first thought appear that any possible improvements will be limited, but we have still gone a long way, at least for the Android ecosystem. I will now briefly discuss some possible extensions of this dissertation.

The Android Ecosystem that we examined was more or less centered around Google Play. A straightforward extension of this dissertation is to study the Android Ecosystems centered around other application stores. Different application stores operate with different properties. For example, the Amazon App Store appears to have a more stringent approval process, potentially using manual vetting, and so may improve the overall security of the ecosystem. As another example, numerous third-party application stores operate in China; many of them do not provide as strong safety as the major stores in the US and their properties may also differ from those of the US

stores simply because of a different demographic. Some of the chapters, specifically AppsPlayground and AutoCog, involve extensive measurements over applications, and thus comparisons of different ecosystems around various application stores are relevant here.

It may also be worthwhile examining if similar solutions are possible in other operating system ecosystems, such as iOS and Windows. These other operating system ecosystems are not as open as Android and so it may not be possible to deploy some solutions proposed here. Moreover, the implementation aspects on different operating systems may also be completely different.

Even on Android, it is possible to develop many more solutions that deal with different parts of the ecosystem, some of which are not even discussed in this dissertation. One example is advertisements and ad networks. Advertisements largely form the monetary backbone of the mobile ecosystems but at the same time have been associated with privacy issues and concerns relating to spreading malware. It will be worth examining and proposing solutions for these parts of the ecosystem as well. It is likely that some of the core techniques described in this dissertation will prove useful for these future solutions and studies. For example, AppsPlayground may be used to run Android applications automatically on a large scale so as to extract and analyze advertisements from them.

References

[1] Android-apktool: A tool for reengineering Android apk files. http://code.google.com/p/android-apktool/. (pages 69 and 135)

[2] Android.Basebridge — Symantec. http://www.symantec.com/security_response/writeup.jsp?docid=2011-060915-4938-99. (page 70)

[3] Android.Bgserv — Symantec. http://www.symantec.com/security_response/writeup.jsp?docid=2011-031005-2918-99. (page 70)

[4] Android.Geinimi — Symantec. http://www.symantec.com/security_response/writeup.jsp?docid=2011-010111-5403-99. (page 70)

[5] AndroidOS.FakePlayer — Symantec. http://www.symantec.com/security_response/writeup.jsp?docid=2010-081100-1646-99. (pages 21 and 70)

[6] Are free Android virus scanners any good? http://www.av-test.org/fileadmin/pdf/avtest_2011-11_free_android_virus_scanner_english.pdf. (pages 52 and 80)

[7] AV-Test. http://www.av-test.org/index.php?L=1. (page 81)

[8] DroidKungFu. http://www.csc.ncsu.edu/faculty/jiang/DroidKungFu.html. (page 74)

[9] Plankton. http://www.csc.ncsu.edu/faculty/jiang/Plankton/. (page 71)

[10] ProGuard. http://proguard.sourceforge.net/. (pages 61, 74, and 81)

[11] Qemu. http://www.qemu.org. (page 26)

[12] Smali: An assembler/disassembler for Android's dex format . http://code.google.com/p/smali/. (pages 69 and 135)

[13] Zelix Klassmaster. http://www.zelix.com/klassmaster/. (pages 74 and 81)

[14] Test: Malware Protection for Android, March 2012. http://www.av-test.org/en/tests/android/. (pages 52 and 80)

[15] Abbot. http://abbot.sourceforge.net/. (page 49)

[16] A. Agrawal and X. Huang. Pairwise statistical significance of local sequence alignment using multiple parameter sets and empirical justification of parameter set change penalty. *BMC bioinformatics*, 10(Suppl 3):S1, 2009. (page 82)

[17] A. Agrawal and X. Huang. Pairwise statistical significance of local sequence alignment using sequence-specific and position-specific substitution matrices. *IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB)*, 8(1):194–205, 2011. (page 82)

[18] L. Apfelbaum and J. Doyle. Model Based Testing. In *Software Quality Week Conference*, pages 296–300, 1997. (page 50)

[19] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *ACM PLDI*, 2014. (pages 140, 149, and 150)

[20] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie. Pscout: analyzing the android permission specification. In *ACM CCS*, 2012. (pages 90, 105, and 117)

[21] AutoIt. http://www.autoitscript.com/site/autoit/. (page 49)

[22] R. Balebako, J. Jung, W. Lu, L. F. Cranor, and C. Nguyen. Little brothers watching you: Raising awareness of data leaks on smartphones. In *Proceedings of the Ninth Symposium on Usable Privacy and Security*, page 12. ACM, 2013. (page 150)

[23] U. Bayer, P. Comparetti, C. Hlauschek, C. Kruegel, and E. Kirda. Scalable, behavior-based malware clustering. In *Network and Distributed System Security Symposium (NDSS)*, 2009. (page 149)

[24] J. Bell and G. E. Kaiser. Phosphor: Illuminating dynamic data flow in the jvm. In *OOPSLA*, 2014. (pages 120, 121, 125, 128, 147, and 149)

[25] K. Benton, L. J. Camp, and V. Garg. Studying the effectiveness of android application permissions requests. In *IEEE PERCOM Workshops*, 2013. (page 117)

[26] A. Beresford, A. Rice, N. Skehin, and R. Sohan. Mockdroid: trading privacy for application functionality on smartphones. In *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications (HotMobile)*, 2011. (page 150)

[27] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, et al. The dacapo benchmarks: Java benchmarking development and analysis. In *ACM Sigplan Notices*, volume 41, pages 169–190. ACM, 2006. (page 144)

[28] A. Bose, X. Hu, K. G. Shin, and T. Park. Behavioral detection of malware on mobile handsets. In *Proceedings of the 6th international conference on Mobile systems, applications, and services*, MobiSys '08. ACM, 2008. (page 83)

[29] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, and B. Shastry. Towards taming privilege-escalation attacks on android. In *NDSS Symposium*, 2012. (page 117)

[30] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani. Crowdroid: behavior-based malware detection system for android. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, pages 15–26. ACM, 2011. (page 82)

[31] D. Chandra and M. Franz. Fine-grained information flow analysis and enforcement in a java virtual machine. In *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*, 2007. (page 149)

[32] P. H. Chia, Y. Yamamoto, and N. Asokan. Is this app safe?: a large scale study on application permissions and risk signals. In *ACM WWW*, 2012. (page 115)

[33] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing inter-application communication in android. In *ACM MobiSys*, 2011. (page 117)

[34] M. Christodorescu and S. Jha. Testing malware detectors. In *Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, ISSTA '04. ACM, 2004. (pages 51 and 80)

[35] M. Christodorescu, S. Jha, and C. Kruegel. Mining specifications of malicious behavior. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC-FSE '07. ACM, 2007. (page 82)

[36] M. Christodorescu, S. Jha, S. Seshia, D. Song, and R. Bryant. Semantics-aware malware detection. In *Security and Privacy, 2005 IEEE Symposium on*, pages 32–46. IEEE, 2005. (pages 77, 78, and 81)

[37] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, Jan. 1999. (page 49)

[38] CNET, February 2013. `http://news.cnet.com/8301-1035_3-57569402-94/android-ios-combine-for-91-percent-of-market/`. (page 51)

[39] 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. (page 81)

[40] C. Collberg, C. Thomborson, and D. Low. Breaking abstractions and unstructuring data structures. In *Computer Languages, 1998. Proceedings. 1998 International Conference on*, pages 28–38. IEEE, 1998. (page 81)

[41] C. Collberg, C. Thomborson, and D. Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Conference Record of the Acm Symposium on Principles of Programming Languages*, volume 25, pages 184–196. ACM, 1998. (page 81)

[42] B. Davis and H. Chen. "retroskeleton: Retrofitting android apps". In *Mobisys*, 2013. (pages 126, 138, and 150)

[43] B. Davis, B. Sanders, A. Khodaverdian, and H. Chen. I-arm-droid: A rewriting framework for in-app reference monitors for android applications. *IEEE Mobile Security Technologies (MoST), San Francisco, CA*, 2012. (pages 126 and 150)

[44] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach. Quire: Lightweight provenance for smart phone operating systems. In *USENIX Security Symposium*, 2011. (page 117)

[45] Q. Do, D. Roth, M. Sammons, Y. Tu, and V. Vydiswaran. Robust, light-weight approaches to compute lexical similarity. *Computer Science Research and Technical Reports, University of Illinois*, 2009. (page 94)

[46] M. Egele, C. Kruegel, E. Kirda, and G. Vigna. Pios: Detecting privacy leaks in ios applications. In *Proceedings of the Network and Distributed System Security Symposium*, 2011. (pages 21 and 149)

[47] M. Egele, C. Kruegel, E. Kirda, H. Yin, and D. Song. Dynamic spyware analysis. In *Usenix Annual Technical Conference*, 2007. (page 149)

[48] W. Enck, P. Gilbert, B. Chun, L. Cox, J. Jung, P. McDaniel, and A. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *USENIX OSDI*, 2010. (pages 21, 22, 31, 89, 120, 121, 125, 128, and 149)

[49] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A study of android application security. In *USENIX Security*, 2011. (pages 21 and 149)

[50] W. Enck, M. Ongtang, and P. McDaniel. On lightweight mobile phone application certification. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 235–245. ACM, 2009. (page 150)

[51] F-Secure. Mobile Threat Report Q3 2012. http://www.f-secure.com/static/doc/labs_global/Research/Mobile%20Threat%20Report%20Q3%202012.pdf. (page 51)

[52] A. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner. A survey of mobile malware in the wild. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, pages 3–14. ACM, 2011. (pages 83 and 85)

[53] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. In *ACM CCS*, 2011. (page 117)

[54] A. P. Felt, S. Egelman, and D. Wagner. I've got 99 problems, but vibration ain't one: A survey of smartphone users' concerns. In *ACM SPSM*, 2012. (pages 89 and 103)

[55] A. P. Felt, K. Greenwood, and D. Wagner. The effectiveness of application permissions. In *USENIX WebApps*, 2011. (pages 115 and 117)

[56] A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner. Android permissions: User attention, comprehension, and behavior. In *ACM SOUPS*, 2012. (pages 84, 89, and 115)

[57] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin. Permission re-delegation: Attacks and defenses. In *USENIX Security Symposium*, 2011. (page 117)

[58] Fortinet. 2012 Threat Predictions. http://blog.fortinet.com/2012-threat-predictions/. (page 52)

[59] M. Fredrikson, S. Jha, M. Christodorescu, R. Sailer, and X. Yan. Synthesizing near-optimal malware specifications from suspicious behaviors. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 45–60. IEEE, 2010. (page 82)

[60] A. Fuchs, A. Chaudhuri, and J. Foster. Scandroid: Automated security certification of android applications. *Manuscript, Univ. of Maryland, http://www.cs.umd.edu/~avik/projects/scandroidascaa*, 2009. (pages 127 and 150)

[61] E. Gabrilovich and S. Markovitch. Computing semantic relatedness using wikipedia-based explicit semantic analysis. In *IJCAI*, 2007. (pages 86 and 95)

[62] C. Gibler, J. Crussell, J. Erickson, and H. Chen. Androidleaks: Automatically detecting potential privacy leaks in android applications on a large scale. *Trust and Trustworthy Computing*, pages 291–307, 2012. (page 149)

[63] Good. Mobile app containerization. http://www1.good.com/secure-mobility-solution/mobile-application-containerization. (page 126)

[64] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang. Riskranker: scalable and accurate zero-day android malware detection. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*, MobiSys '12. ACM, 2012. (page 82)

[65] V. Haldar, D. Chandra, and M. Franz. Dynamic taint propagation for java. In *21st Annual Computer Security Applications Conference (ACSAC)*. IEEE, 2005. (page 149)

[66] W. Han, Z. Fang, L. T. Yang, G. Pan, and Z. Wu. Collaborative policy administration. *IEEE TPDS*, 25(2):498–507, 2014. (page 117)

[67] S. Hao, D. Li, W. G. Halfond, and R. Govindan. Sif: a selective instrumentation framework for mobile applications. In *Proceeding of the 11th annual international conference on Mobile systems, applications, and services*, pages 167–180. ACM, 2013. (page 138)

[68] L. Harris and B. Miller. Practical analysis of stripped binary code. *ACM SIGARCH Computer Architecture News*, 33(5):63–68, 2005. (page 78)

[69] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall. "these arent the droids youre looking for": Retrofitting android to protect data from imperious applications. 2011. (pages 42, 46, and 89)

[70] C. Hu and I. Neamtiu. Automating gui testing for android applications. In *Proceeding of the 6th international workshop on Automation of software test*, 2011. (page 50)

[71] Y. Huang, S. Huang, T. Lin, and C. Tsai. Web application security assessment by fault injection and behavior monitoring. In *Proceedings of the 12th international conference on World Wide Web*, pages 148–159, 2003. (page 39)

[72] X. Jiang and X. Zhu. veye: behavioral footprinting for self-propagating worm detection and profiling. *Knowledge and information systems*, 18(2):231–262, 2009. (page 82)

[73] J. P. John, A. Moshchuk, S. D. Gribble, and A. Krishnamurthy. Studying spamming botnets using Botlab. In *Proceedings of the 6th USENIX symposium on Networked systems design and implementation*, pages 291–306, Berkeley, CA, USA, 2009. USENIX Association. (page 49)

[74] J. Jung, A. Sheth, B. Greenstein, D. Wetherall, G. Maganis, and T. Kohno. Privacy oracle: a system for finding application leaks with black box differential testing. In *CCS '08: Proceedings of the 15th ACM conference on Computer and communications security*, pages 279–288, New York, NY, USA, 2008. ACM. (page 49)

[75] M. Kang, S. McCamant, P. Poosankam, and D. Song. Dta++: Dynamic taint analysis with targeted control-flow propagation. *Proc. of the 18th NDSS*, 2011. (page 147)

[76] M. G. Kendall. Rank correlation methods. 1948. (page 113)

[77] K. Kennedy, E. Gustafson, and H. Chen. Quantifying the effects of removing permissions from android applications. In *IEEE MoST*, 2013. (page 117)

[78] H. Kim, J. Smith, and K. G. Shin. Detecting energy-greedy anomalies and mobile malware variants. In *Proceedings of the 6th international conference on Mobile systems, applications, and services*, MobiSys '08. ACM, 2008. (page 83)

[79] J. Kim, Y. Yoon, K. Yi, J. Shin, and S. Center. Scandal: Static analyzer for detecting privacy leaks in android applications. In *Mobile Security Technologies (MoST), Workshop on*, 2012. (page 150)

[80] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976. (page 49)

[81] T. Kiss and J. Strunk. Unsupervised multilingual sentence boundary detection. *Computational Linguistics*, 32(4):485–525, 2006. (page 92)

[82] C. Kolbitsch, P. Comparetti, C. Kruegel, E. Kirda, X. Zhou, and X. Wang. Effective and efficient malware detection at the end host. In *Proceedings of the 18th conference on USENIX security symposium*, pages 351–366. USENIX Association, 2009. (page 82)

[83] R. Komondoor and S. Horwitz. Semantics-preserving procedure extraction. In *In POPL*, pages 155–169. ACM Press, 2000. (page 64)

[84] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna. Static disassembly of obfuscated binaries. In *Proceedings of the 13th USENIX Security Symposium*, pages 255–270, 2004. (page 82)

[85] C. Lim, N. Singh, and S. Yajnik. A log mining approach to failure analysis of enterprise telephony systems. In *IEEE DSN*, 2008. (page 116)

[86] J. Lin, S. Amini, J. I. Hong, N. Sadeh, J. Lindqvist, and J. Zhang. Expectation and purpose: understanding users' mental models of mobile app privacy through crowdsourcing. In *ACM Ubicomp*, 2012. (page 117)

[87] J. Lin, S. Amini, J. I. Hong, N. Sadeh, J. Lindqvist, and J. Zhang. Expectation and purpose: understanding users' mental models of mobile app privacy through crowdsourcing. In *Proceedings of the 2012 ACM Conference on Ubiquitous Computing*, pages 501–510. ACM, 2012. (page 150)

[88] C. Linn and S. Debray. Obfuscation of executable code to improve resistance to static disassembly. In *Proceedings of the 10th ACM conference on Computer and communications security*, pages 290–299. ACM, 2003. (page 78)

[89] L. Liu, G. Yan, X. Zhang, and S. Chen. Virusmeter: Preventing your cellphone from spies. In *Recent Advances in Intrusion Detection*, pages 244–264. Springer, 2009. (page 83)

[90] H. Lockheimer. Android and security, February 2012. `http://googlemobile.blogspot.com/2012/02/android-and-security.html`. (page 79)

[91] Lookout. Geinimi Trojan Technical Analysis. `http://blog.mylookout.com/blog/2011/01/07/geinimi-trojan-technical-analysis/`. (page 74)

[92] Lookout. Update: Security Alert: DroidDream Malware Found in Official Android Market. `http://blog.mylookout.com/blog/2011/03/01/security-alert-malware-found-in-official-android-market-droiddream/`. (pages 21 and 70)

[93] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang. CHEX: statically vetting Android apps for component hijacking vulnerabilities. In *Proceedings of the 2012 ACM conference on Computer and communications security*, 2012. (pages 127 and 150)

[94] McAfee. McAfee Threats Report: Third Quarter 2011. `http://www.mcafee.com/us/resources/reports/rp-quarterly-threat-q3-2011.pdf`. (page 51)

[95] A. Memon. An event-flow model of gui-based applications for testing. *Software Testing, Verification and Reliability*, 17(3):137–157, 2007. (page 50)

[96] A. Memon, I. Banerjee, and A. Nagarajan. GUI Ripping: Reverse Engineering of Graphical User Interfaces for Testing. *Reverse Engineering, Working Conference on*, pages 260+, 2003. (pages 23, 38, 45, and 50)

[97] A. M. Memon, M. E. Pollack, and M. L. Soffa. Hierarchical GUI test case generation using automated planning. *IEEE Transactions on Software Engineering*, 27(2):144–155, Feb. 2001. (page 50)

[98] Microsoft. Watch out for fake virus alerts. `http://www.microsoft.com/security/pc-security/antivirus-rogue.aspx`. (page 79)

[99] R. Mihalcea, C. Corley, and C. Strapparava. Corpus-based and knowledge-based measures of text semantic similarity. In *AAAI*, 2006. (page 95)

[100] MobileIron. Appconnect. `http://www.mobileiron.com/en/products/appconnect`. (page 126)

[101] Y. Nadji, J. Giffin, and P. Traynor. Automated remote repair for mobile malware. In *Proceedings of the 27th Annual Computer Security Applications Conference*, pages 413–422. ACM, 2011. (page 83)

[102] S. Nanda, W. Li, L. Lam, and T. Chiueh. Bird: Binary interpretation using runtime disassembly. In *Code Generation and Optimization, 2006. CGO 2006. International Symposium on*, pages 12–pp. IEEE, 2006. (page 78)

[103] M. Nauman, S. Khan, and X. Zhang. Apex: extending android permission model and enforcement with user-defined runtime constraints. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, pages 328–332. ACM, 2010. (page 150)

[104] S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443–453, March 1970. (page 82)

[105] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *NDSS*, 2005. (pages 127 and 149)

[106] J. Oberheide. Dissecting android's bouncer, June 2012. `https://blog.duosecurity.com/2012/06/dissecting-androids-bouncer/`. (pages 48 and 79)

[107] D. L. Olson and D. Delen. *Advanced data mining techniques*. Springer, 2008. (page 98)

[108] R. Pandita, X. Xiao, W. Yang, W. Enck, and T. Xie. Whyper: Towards automating risk assessment of mobile applications. In *USENIX Security*, 2013. (pages 85, 90, 103, 106, 112, and 117)

[109] R. Pandita, X. Xiao, H. Zhong, T. Xie, S. Oney, and A. Paradkar. Inferring method specifications from natural language api descriptions. In *IEEE ICSE*, 2012. (page 117)

[110] M. Parkour. Contagio Mobile. Mobile Malware Mini Dump. `http://contagiominidump.blogspot.com/`. (page 70)

[111] H. Peng, C. Gates, B. Sarma, N. Li, Y. Qi, R. Potharaju, C. Nita-Rotaru, and I. Molloy. Using probabilistic generative models for ranking risks of android apps. In *ACM CCS*, 2012. (page 82)

[112] R. Potharaju, N. Jain, and C. Nita-Rotaru. Juggling the jigsaw: Towards automated problem inference from network trouble tickets. In *USENIX NSDI*, 2013. (page 116)

[113] M. D. Preda, M. Christodorescu, S. Jha, and S. Debray. A semantics-based approach to malware detection. In *Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '07. ACM, 2007. (page 82)

[114] A. Pretschner, O. Slotosch, E. Aiglstorfer, and S. Kriebel. Model-based testing for real. *International Journal on Software Tools for Technology Transfer (STTT)*, 5(2):140–157, Mar. 2004. (page 50)

[115] F. Qin, C. Wang, Z. Li, H. Kim, Y. Zhou, and Y. Wu. Lift: A low-overhead practical information flow tracking system for detecting security attacks. In *Microarchitecture, 2006. MICRO-39. 39th Annual IEEE/ACM International Symposium on*, 2006. (page 149)

[116] T. Qiu, Z. Ge, D. Pei, J. Wang, and J. Xu. What happened in my network: mining network events from router syslogs. In *ACM SIGCOMM*, 2010. (page 116)

[117] T. Raffetseder, C. Kruegel, and E. Kirda. Detecting system emulators. *Information Security*, pages 1–18, 2007. (page 33)

[118] S. Raghavan and H. Garcia-Molina. Crawling the hidden web. In *Proceedings of the International Conference on Very Large Data Bases*, pages 129–138, 2001. (page 39)

[119] J. Raphael. Exclusive: Inside android 4.2's powerful new security system, November 2012. `http://blogs.computerworld.com/android/21259/android-42-security`. (page 79)

[120] V. Rastogi, Y. Chen, and W. Enck. AppsPlayground: Automatic Security Analysis of Smartphone Applications. In *Proceedings of ACM CODASPY 2013*, February 2013. (pages 121 and 142)

[121] L. Ravindranath, J. Padhye, S. Agarwal, R. Mahajan, I. Obermiller, and S. Shayandeh. Appinsight: Mobile app performance monitoring in the wild. In *OSDI*, pages 107–120, 2012. (page 150)

[122] J. C. Reynar and A. Ratnaparkhi. A maximum entropy approach to identifying sentence boundaries. In *Proceedings of the fifth conference on Applied natural language processing*, 1997. (page 92)

[123] Robotium. `http://code.google.com/p/robotium/`. (page 49)

[124] N. J. Rubenking. PCMag. The Best Antivirus for 2012. `http://www.pcmag.com/article2/0,2817,2372364,00.asp`. (page 81)

[125] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A symbolic execution framework for javascript. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 513–528. IEEE, 2010. (page 34)

[126] P. Saxena, R. Sekar, and V. Puranik. Efficient fine-grained binary instrumentation with applications to taint-tracking. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, pages 74–83. ACM, 2008. (pages 78 and 150)

[127] E. Schwartz, T. Avgerinos, and D. Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Security and Privacy (SP), 2010 IEEE Symposium on*, 2010. (pages 128, 147, and 149)

[128] B. Schwarz, S. Debray, and G. Andrews. Disassembly of executable code revisited. In *Reverse Engineering, 2002. Proceedings. Ninth Working Conference on*, pages 45–54. IEEE, 2002. (page 78)

[129] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. *SIGSOFT Softw. Eng. Notes*, 30(5):263–272, Sept. 2005. (page 49)

[130] M. Sharif, A. Lanzi, J. Giffin, and W. Lee. Automatic reverse engineering of malware emulators. In *Security and Privacy, 2009 30th IEEE Symposium on*, 2009. (page 149)

[131] A. Slowinska, T. Stancescu, and H. Bos. Body armor for binaries: preventing buffer overflows without recompilation. In *Proceedings of the USENIX Annual Technical Conference*, 2012. (page 150)

[132] T. F. Smith and M. S. Waterman. Identification of Common Molecular Subsequences. *Journal of Molecular Biology*, 147(1):195–197, 1981. (page 82)

[133] R. Socher, J. Bauer, C. D. Manning, and A. Y. Ng. Parsing with compositional vector grammars. In *Proceedings of the ACL*, 2013. (pages 92 and 112)

[134] G. Suh, J. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *ACM SIGPLAN Notices*, volume 39, pages 85–96, 2004. (page 149)

[135] Symantec. Server-side Polymorphic Android Applications. `http://www.symantec.com/connect/blogs/server-side-polymorphic-android-applications`. (page 52)

[136] Y.-M. Wang, D. Beck, X. Jiang, and R. Roussev. Automated Web Patrol with Strider HoneyMonkeys: Finding Web Sites that Exploit Browser Vulnerabilities. In *IN NDSS*, 2006. (page 48)

[137] R. Whitwam. Circumventing Google's Bouncer, Android's anti-malware system, June 2012. http://www.extremetech.com/computing/130424-circumventing-googles-bouncer-androids-anti-malware-system. (pages 48 and 79)

[138] C. Willems, T. Holz, and F. Freiling. Toward Automated Dynamic Malware Analysis Using CWSandbox. *IEEE Security and Privacy*, 5(2):32–39, Mar. 2007. (page 48)

[139] G. Wondracek, P. M. Comparetti, C. Kruegel, E. Kirda, and S. S. S. Anna. Automatic network protocol analysis. In *15th Symposium on Network and Distributed System Security (NDSS)*, 2008. (page 82)

[140] R. Xu, H. Saıdi, and R. Anderson. Aurasium: Practical policy enforcement for android applications. In *Proceedings of the 21st USENIX conference on Security*, 2012. (pages 126 and 150)

[141] W. Xu, S. Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: a practical approach to defeat a wide range of attacks. In *Proceedings of the 15th conference on USENIX Security Symposium*, 2006. (pages 127 and 150)

[142] L.-K. Yan and H. Yin. DroidScope: Seamlessly Reconstructing the OS and Dalvik. In *Proceedings of USENIX Security Symposium*. USENIX Association, 2012. (pages 21, 28, 48, and 89)

[143] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM conference on Computer and communications security*, 2007. (page 149)

[144] M. Zhang and H. Yin. Appsealer: Automatic generation of vulnerability-specific patches for preventing component hijacking attacks in android applications. In *NDSS*, 2014. (page 150)

[145] M. Zhang and H. Yin. Efficient, context-aware privacy leakage confinement for android applications without firmware modding. In *ASIACCS*, 2014. (page 150)

[146] C. Zheng, S. Zhu, S. Dai, G. Gu, X. Gong, X. Han, and W. Zou. Smartdroid: an automatic system for revealing ui-based trigger conditions in android applications. In *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices*, pages 93–104. ACM, 2012. (page 49)

[147] M. Zheng, P. Lee, and J. Lui. Adam: An automatic and extensible platform to stress test android anti-virus systems. *DIMVA*, July 2012. (page 80)

[148] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. *Security and Privacy, IEEE Symposium on*, 2012. (pages 34 and 83)

[149] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang. Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. In *NDSS*, 2012. (pages 82 and 85)

[150] Y. Zhou, X. Zhang, X. Jiang, and V. Freeh. Taming information-stealing smartphone applications (on android). *Trust and Trustworthy Computing*, pages 93–107, 2011. (page 150)

[151] D. Zhu, J. Jung, D. Song, T. Kohno, and D. Wetherall. Tainteraser: protecting sensitive data leaks using application-level taint tracking. *ACM SIGOPS Operating Systems Review*, 45(1):142–154, 2011. (pages 127 and 149)