Tracking Explicit and Control Flows in Java and Native Android Apps Code

Mariem Graa[1], Nora Cuppens-Boulahia[1], Frédéric Cuppens[1] and Jean-Louis Lanet[2]
[1]*Telecom-Bretagne, 2 Rue de la Châtaigneraie, 35576 Cesson Sévigné, France*
[2]*Université de Rennes 1, Campus de Beaulieu, 263 Avenue Général Leclerc, 35042 Rennes, France*

Keywords: Android System, Native Libraries, Dynamic Analysis, Static Analysis, Control Dependencies, Leakage of Sensitive Information, Information Flow Tracking.

Abstract: The native app development is increased in Android systems to implement CPU-intensive applications such as game engines, signal processing, and physics simulation. However, native code analysis is very difficult and requires a lot of time which explains the limited number of systems that track information flow in native libraries. But, none of them detects the sensitive information leakage through control flows at native level. In this paper, we combine dynamic and static taint analysis to propagate taint along control dependencies. Our approach has proven to be effective in analyzing several malicious Android applications that invoke native librairies with reasonable performance overheads.

1 INTRODUCTION

Mobile devices have become a necessity for many people throughout the world. The ability for communication, GPS navigation, web browsing and multimedia entertainment are only a few of the reasons for the increasing use of mobile devices. According to a recent Gartner report (Rob van der Meulen, 2013), 455.6 million of worldwide mobile phones were sold in the third quarter of 2013, which corresponds to 5.7% increase from the same period last year. Sales of smartphones accounted for 55% of overall mobile phone sales in the third quarter of 2013. In particular, Android surpassed 80% market share in the same period.

In order to satisfy Android users' requirements, the development of Android applications have been growing at a high rate. In May 2013, 48 billion apps have been installed from the Google Play store (Warren, 2013). Most of these applications use native libraries to improve performance. Table 1 shows the increase in the number of Android apps employing native libraries from 4.2% in 2011 to 24% in 2013. These applications can invoke System.load() or System.loadLibrary() to load native libraries such as game, music and audio and communication apps, or can contain native libraries without calling these methods or can be written in pure native code. Note that the restricted number of applications in pure native code is due to the limitations of the NDK (Native Developpement Kit) APIs.

Table 1: Increase in the number of Android apps using native libraries.

Period	Percentage of apps using native libraries
May-Jun 2011	4.2% (Zhou et al., 2012)
Sep-Oct 2011	9.42% (Grace et al., 2012)
Jun 2012-Jun 2013	16.46% (Qian et al., 2014)
July 2013-Dec 2013	24% (Spreitzenbarth et al., 2013)

The native libraries can contain sensitive data such as phone identity, user contacts, pictures and locations. An attacker can exploit this native code to get private information.

Dynamic taint analysis can be used for checking information flows through JNI (Java Native Interface) and to control the handling of private data in Android systems (Qian et al., 2014). This technique assigns taint to sensitive data. Then, it tracks propagation of tainted data to detect leakage of private information by malicious applications. This, can be detected when sensitive data are used in a taint sink (network interface).

```
boolean x;
char c[256];
if( gets(c) != user_location )
    x=false;
else
    x=true;
NetworkTransfer (x);
```

Listing 1: Control flow example.

Two types of flows are defined: explicit flows such as $x = y$, where we observe an explicit transfer of a

Graa, M., Cuppens-Boulahia, N., Cuppens, F. and Lanet, J-L.
Tracking Explicit and Control Flows in Java and Native Android Apps Code.
DOI: 10.5220/0005686603070316
In *Proceedings of the 2nd International Conference on Information Systems Security and Privacy (ICISSP 2016)*, pages 307-316
ISBN: 978-989-758-167-0
Copyright © 2016 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

value from x to y, and control flows shown in Listing 1 that occur in the control flow structure of the program. In this example, the attacker tries to get the user location by exploiting control flows. The value of the variable x depends on the condition and it informs the attacker about the value of user location. When the test was positive, x is modified and it is leaked to the network without being detected because the dynamic taint analysis approach implemented in the Android native code does not propagate taint along control dependencies. Thus, malicious applications can get privacy sensitive data through control flows in native code. Our objective is to detect private information leakage by untrusted smartphone applications exploiting control flows in native code. In this context, many challenges need to be addressed:

- The resource constrained nature of Smartphones: the limited processing and memory capacity of smartphones make it difficult to use information flow tracking systems (Yin et al., 2007), (Clause et al., 2007), (Kang et al., 2011), (Song et al., 2008). Thus, it is necessary to design a resource efficient security mechanism.
- The unavailable application source code: the source code of smartphone applications is often unavailable.
- False negatives: while the most used information flow tracking systems (Enck et al., 2010), (Hornyack et al., 2011), (Qian et al., 2014), (Yan and Yin, 2012) do not track control flows, false negatives could occur and cause security flaws.

The approach cited in (Graa et al., 2014) proposed an enhancement of dynamic taint analysis that propagates taint along control dependencies in Java Android applications' code. In this paper, we improve this approach by considering also the native code and combining dynamic taint analysis and static analysis to control the manipulation of private data by third-party apps that exploit control flows in native libraries to leak sensitive information. To the best of our knowledge, this is the first paper that tracks control flows in native Android code to detect the leakage of private data. Analysis of applications' behavior at native level is very difficult and requires a lot of time. To be practical, the performance overhead of our system runtime must be reasonable. In addition, sufficient contextual information about propagation of private data in Java and native levels is needed. Thus, we use the data tainting technique and we assign taint to sensitive information. Also, it is not trivial to track control flow in native code. So, we use static analysis to detect control dependencies.

This paper is organized as follows: section 2 defines the problem statement. Related works about existing systems that consider native libraries invoked by Android applications and existing approaches that detect control flows are discussed in section 3. We give an overview of our solution based on a hybrid approach that improves the functionality of NDroid by considering the control dependencies in section 4. Section 5 describes implementation details of our approach. We test the effectiveness of our approach by analysing several Android applications and we study our approach taint tracking overhead in section 6. We discuss the limitations of our work in section 7. Finally, section 8 concludes.

2 PROBLEM STATEMENT

We present in this section the problem that our approach solves.

The under-tainting problem occurs when some values should be marked as tainted, but are not. The dynamic taint analysis approaches implemented in Android native code do not propagate taint in control flows which can cause an under-tainting problem. Consider the example in Listing 3 that presents an under tainting problem in Android native code. The malicious program written in Java code (see Listing 2) gets the device id and passes it as argument to a native function (written in C). In the native code presented in Listing 3, the attacker compares each character of private data with symbols in *AsciiTable* (*TabAsc*). Then, he stores the sequence of characters found in the string Z. The variable Z contains the value of the private data but it is not tainted using existing dynamic analysis taint systems. TaintDroid taints the returned value of a JNI function if one argument is tainted. In this case, the native function does not return a value.

```
package com.tuto.attackndk;
public class MainActivity extends Activity {
  static {
    System.loadLibrary("attackndk");
  }
  public static native void invokeNativeFunction
     (String IMEI);
  @Override
    protected void onCreate(Bundle
     savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
          String device_id = GetDeviceId();
            invokeNativeFunction(device_id);
      }
}
```

Listing 2: Attack exploiting native code based on control dependencies.

The approach presented in (Graa et al., 2014) does not propagate taint in native code. NDroid under-taints the control flows of native code. Thus, variable Z is leaked through JNI without any warning reports. In this example, the taint sink (network interface) is defined in the native code. So, the variable Z is leaked

through the network connection ($Send_Data(Z)$). The approaches cited in (Enck et al., 2010), (Graa et al., 2014) do not implement taint sink in the native code. But, NDroid instruments native libraries to define the taint sink. As the variable Z sent through the network connection is not tainted, the leakage of this data cannot be detected.

```
#include <string.h>
#include <jni.h>
void Java_com_tuto_attackndk_MainActivity_
invokeNativeFunction(JNIEnv* env, jobject thiz,
    jstring IMEI)
{
 String Private_Data;
 String Z;
  strcpy(Private_Data, IMEI);
    for(int i = 0; i < sizeof(Private_Data); i
    ++)
          {
        char s;
        sprintf(s, "%d", i);
        for(int j = 1; j < sizeof(TabAsc); j++)
            if (strcmp(s,TabAsc[j]) == 0)
                strcat(Z,TabAsc[j]);
      }
      Send_Data(Z);
}
```

Listing 3: Native malicious function.

The taint sink can be defined in the Java code. In this case, another Java program searches the sensitive information from the native code, or the native method calls and passes the sensitive information to Java code. Then, the Java code leaks the private data. As it is not tainted, it will be sent without any warning reports.

As the dynamic taint analysis approaches cannot detect a leakage of sensitive information caused by the under-tainting problem. We aim to enhance these approaches by tracking control flow in the Android native code. To do so, we also use static analysis to detect control dependencies. The dynamic analysis allows tainting variables to which a value is assigned in the conditional instruction exploiting information provided by the static analysis.

We present existing systems that track information flows in native libraries invoked by Android applications and existing approaches that detect control flows in the next section.

3 RELATED WORK

Static Analysis Approaches. Static analysis is used to detect leakage of sensitive data. ComDroid (Chin et al., 2011), SCANDROID (Fuchs et al., 2009), Ded (Enck et al., 2011) and FLOWDROID (Arzt et al., 2014) allow a static analysis of third party Android applications' code. One of the limits of these static analysis approaches is that they cannot capture all runtime configurations and inputs.

Dynamic Analysis Approaches. TaintDroid (Enck et al., 2010) implements dynamic taint analysis to determine when privacy sensitive information leaves the phone. TaintDroid defines taint propagation logic only for explicit (direct) flows and does not track indirect flows (control flows). For native code, TaintDroid adds an additional propagation heuristic patch that assigns the union of the native method argument taint tags to the taint tag of the return value. This, does not reflect the exact taint propagation in native code, because it under-taints explicit information flows from native code to Dalvik virtual machine (DVM). AppFence (Hornyack et al., 2011) extends Taintdroid to implement policy enforcement. A significant limitation of these dynamic taint analysis approaches implemented in Android systems is that they track only explicit flows in Java code level and they do not consider native libraries.

Native Libraries Approaches. Fedler *et al.* (Fedler et al., 2013) assert that all current local root exploits are exclusively implemented as native code and can be dynamically downloaded and runned by any app. Since, the lack of control mechanisms for the execution of native code poses a major threat to the security of Android devices, Fedler *et al.* propose mechanisms to control native code execution. Some works have been undertaken to consider native libraries in Android applications. DroidRanger (Zhou et al., 2012) records any calls to the Android framework APIs for the dynamically loaded Java code and their arguments to provide rich semantic information about an app's behavior. For the dynamically-loaded native code, it collects system calls made by the native code using a kernel module that catches the system calls table in the (Linux) kernel. RiskRanker (Grace et al., 2012) compares native code of apps with the signatures of known root exploits. To analyze potential harmful applications for the Android platform, AASandbox (Blasing et al., 2010) realizes system and library call monitoring. Paranoid (Portokalidis et al., 2010) and Crowdroid (Burguera et al., 2011) intercept system calls and signals of processes. The dynamic analyzer presented in (Spreitzenbarth et al., 2013) traces code included in native shared objects, those included with the app through the NDK as well as those shipped with Android by intercepting library calls of a monitored application. CopperDroid (Reina et al., 2013) instruments the Android emulator to enable system call tracking and support an out-of-the-box system call-centric analysis. DroidScope (Yan and Yin, 2012), an emulation-based Android malware analysis engine, used to analyze the Java and native components of Android Applications. It implements

several analysis tools to collect detailed native and Dalvik instruction traces, profile API-level activity, and track information leakage through both the Java and native components using taint analysis. One of the limits of DroidScope is that it incurs high overhead. DROIT (WANG and SHIEH, 2015) is an emulation-based taint tracking system. It tracks data flow at Java object level and switches to instruction level instrumentation when native code starts to take over. DROIT is designed to profile the program behavior. So the program is considered as the taint source. On the contrary, in TaintDroid the data in the program presents the taint sources. NDroid (Qian et al., 2014) extends TaintDroid to track explicit flow in native code. The major disadvantage of all approaches that consider native libraries is that they do not propagate taint in control flows.

Control Flows Approaches. Cavallaro *et al.* (Cavallaro et al., 2008) describe the evasion techniques that can easily defeat dynamic information flow analysis using control dependencies. They show that it is necessary to reason about assignments that take place on the program branches. We implement the same idea in our taint propagation rules.
Some approaches exist in the literature to track control flows (Clause et al., 2007), (Egele et al., 2007), (Song et al., 2008), (Kang et al., 2011), (Nair et al., 2008). They combine static and dynamic taint analysis techniques to correctly identify control flows and to detect a leak of sensitive information. Fenton (Fenton, 1974) defined a Data Mark Machine, an abstract model, to handle control flows. Fenton assigns a security class to data and defines an interaction matrix to manipulate information in the system. Aries (Brown and Knight Jr, 2001) considers that writing to a particular location within a branch is disallowed when the security class associated with that location is equal to or less restrictive than the security class of the program counter. The Aries approach is based only on high and low security classes. Denning (Denning, 1975) enhances the run time mechanism used by Fenton with a compile time mechanism to handle all branches in the conditional structure. Denning inserts updating instructions whether the branch is taken or not. This reflects clearly the information flow. Furthermore, these approaches are not implemented in smartphones application. Our approach is inspired from these prior works, but addresses different challenges specific to mobile phones like resource limitations.

The approach cited in (Graa et al., 2014), (Graa et al., 2012) extended TaintDroid and combined dynamic taint analysis and static analysis to track control flows at Java level in Android operating system. A proof of the correctness and completeness of this approach is presented in (Graa et al., 2013). However, this approach does not consider native libraries.

We draw our inspiration from the prior works and we enhance the approach presented in (Graa et al., 2014) by propagating taint along control flow in native code. We describe our approach in more details in the following section.

4 SYSTEM DESIGN

The approach presented in (Graa et al., 2014) allows tracking information flows (explicit and control flows) in Java Android applications to detect leakage of sensitive data. It enhances the TaintDroid system based on dynamic taint analysis to control the manipulation of private data by third party Android applications. First, authors assign taint to sensitive data. Then, they track propagation of tainted data in the Java components. They issue warning reports when the tainted data are leaked by malicious applications.

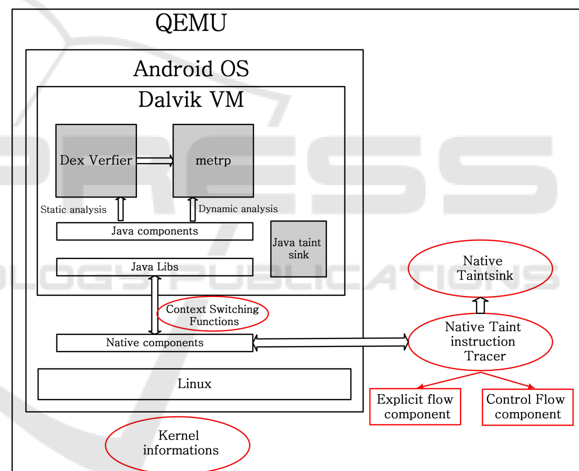

Figure 1: Modified architecture to handle control flow in native code.

As TaintDroid cannot detect control flows, they combine the static and dynamic taint analyses to solve the under tainting problem. The static analysis is used to detect control dependencies and to have an overview of all conditional branches in the program. They use information provided by static analysis to propagate taint along all control dependencies in the dynamic analysis phase.

Components colored in gray (see Figure 1) present the Java Taint Instruction Tracer module of this approach (Graa et al., 2014), (Graa et al., 2012) implemented in the Dalvik VM for tracking information flows at Java level. Authors modify the Dex verifier to statically analyze the Java Android app's code and the interpreter for the dynamic analysis. The Java taint sink is defined by instrumenting the Java framework

libraries. For more implementation details of the Java Taint Instruction Tracer module, please refer to (Graa et al., 2012), (Graa et al., 2014).

For JNI methods, authors patched the call bridge to provide taint propagation. In addition, they defined a method profile that is a list of (from; to) pairs indicating flows between variables for tag propagation in JNI methods. It will be used to update taints when a JNI method returns. Authors add an additional propagation heuristic patch that assigns taint to the returned value of a JNI function if one parameter is tainted. This, does not reflect the exact taint propagation in native libraries.

Thus, to ensure the proper propagation of the native code, we implement the Native Taint Instruction Tracer module (see Figure 1) that instruments the ARM/Thumb instructions. Also, we use static analysis to enumerate the information flows (control flows) for all JNI methods.

In addition, when a native function is invoked in (Graa et al., 2014), the corresponding taint is not stored in the native runtime stack. So, they do not get and set taints correctly when the context (Java and native) switches. Therefore, we define the Context Switching Functions module (see Figure 1). This module instruments JNI-related functions, through which information flows cross the boundary between the Java context and the native context to maintain taints between the two contexts.

Moreover, we implement the Native Taint sink module (see Figure 1) which hooks a selected system call. Finally, we get Android Linux kernel information of processes and memory maps (Information Kernel module). We detail these modules in the following section.

5 HANDLING CONTROL FLOW IN NATIVE LIBRAIRIES

The architecture of our approach is presented in Figure 1. The Android system runs on top of an emulator QEMU (Wiki, 2015) which provides information about all generated ARM/Thumb Android instructions. Thus, we made a number of modifications to QEMU to implement our proposed approach. We define four modules: (1) Native Taint Instruction Tracer module that handles explicit and control flows at native level; (2) Context Switching Functions module; (3) Native Taint sink module and (4) Information Kernel module.

5.1 Native Taint Instruction Tracer

The Native Taint Instruction Tracer module is composed of (1) Explicit flow component that tracks direct flows and (2) Control flow component that tracks indirect flows at native level.

We instrument the ARM/Thumb instructions of native libraries invoked by Android applications to propagate taint in explicit flow. We handle unary, binary, move operations. For these operations, we add taint propagation instructions that assign to the destination register (R_d) the combination of all taints of source registers (R_n, R_m). We propagate taint tags in explicit flow referencing data flow rules presented in Table 2. *#imm* represents the immediate number with a null value of taint. The $t(M[addr])$ is the taint of memory at address *addr* that is calculated using R_n and *#imm* ($Cal(R_n, \#imm)$). *LDM* and *STM* represent the load and store instructions. The operator "$\oplus$" is used to combine taints of objects. The operator "$\otimes$" indicates a binary operation between two registers. Listing 4 and Listing 5 present the instrumentation code to handle explicit flow of respectively move and binary operations.

It is more difficult to handle control flows because it is necessary to detect all conditional branches. Thus, to track control flow at native level, we use a static analysis which checks instructions of native methods at load time. This analysis is based on the control flow graphs (Aho et al., 1986; Allen, 1970) which are analyzed to determine branches in the conditional structure. The control flow graph is composed of nodes that represent basic blocks and directed edges that represent jumps in the control flow. A basic block is assigned to each control flow branch. We detect the flow of the condition-dependencies from blocks in the graph using the *BitmapBits* which is an array of bits. Setting all bits indicates that the flow of control is merged and the basic block does not depend on control condition.

```
case I_MOV:

d->Rd = (w >> 8) & b111;
/* CONTROL FLOWS START */
if(context_cond)
{
setRegTaint(d->Rd,(getRegTaint(d->Rn)|taint_cond
      ));
}
/* CONTROL FLOWS END */
else
/* EXPLICIT FLOWS START */
{
setRegToReg(d->Rd, d->Rn);
}
/* EXPLICIT FLOWS END */
return 0;
```

Listing 4: Instrumenting I_MOV instruction to propagate taint in control flows.

When one bit is set, the basic block depends on the control condition. The basic block represents the conditional instruction when no bit is set. Also, we detect variable assignment in a basic block of the control flow graph. When we run the native methods,

Table 2: Explicit Flow Propagation Logic.

Instruction Format	Instruction Semantics	Taint Propagation
mov R_d,#imm	$R_d \leftarrow \#imm$	$Taint(R_d) \leftarrow \emptyset$
mov R_d,R_m	$R_d \leftarrow R_m$	$Taint(R_d) \leftarrow Taint(R_m)$
unary-op R_d,R_m	$R_d \leftarrow \otimes R_m$	$Taint(R_d) \leftarrow Taint(R_m)$
binary-op R_d,R_n,R_m	$R_d \leftarrow R_n \otimes R_m$	$Taint(R_d) \leftarrow Taint(R_n) \oplus Taint(R_m)$
binary-op R_d,R_m	$R_d \leftarrow R_d \otimes R_m$	$Taint(R_d) \leftarrow Taint(R_d) \oplus Taint(R_m)$
binary-op R_d,R_m,#imm	$R_d \leftarrow R_m \otimes \#imm$	$Taint(R_d) \leftarrow Taint(R_m)$
STR R_d,R_n,#imm	$addr \leftarrow Cal(R_n,\#imm)$, $M[addr] \leftarrow R_d$	$Taint(M[addr]) \leftarrow Taint(R_d)$
LDR R_d,R_n,#imm	$addr \leftarrow Cal(R_n,\#imm)$, $R_d \leftarrow M[addr]$	$Taint(R_d) \leftarrow Taint(M[addr]) \oplus Taint(R_n)$
STM(PUSH) regList, R_n,#imm	$startAddress/endAddress \leftarrow Cal(R_n,\#imm)$, $\{M[startAddress],M[endAddress]\} \leftarrow \{R_i,R_j\}$	$Taint(\{M[startAddress],M[endAddress]\}) \leftarrow Taint(\{R_i,R_j\})$
LDM(POP) regList, R_n,#imm	$startAddress/endAddress \leftarrow Cal(R_n,\#imm), \{R_i,R_j\} \leftarrow \{M[startAddress],M[endAddress]\}$	$Taint(\{R_i,R_j\}) \leftarrow Taint(R_n) \oplus Taint(\{M[startAddress],M[endAddress]\})$

we taint these variables if the condition is tainted. To do this, we use dynamic analysis and we instrument third-party native libraries conditional instructions (see Listing 4 and Listing 5 for move and binary instruction in a conditional statement). This analysis uses information provided by the static analysis such as the *BitmapBits* to detect condition-dependencies from block in the graph and variable assignment. Then, we define a context taint as condition taint. We taint modified variables that exist in the conditional instruction according to rules of taint propagation defined and proven in (Graa et al., 2013). If the branch is taken then $Taint(x) = ContextTaint \oplus Taint(explicit flowstatement)$.

```
case T_THUMB_3REG:
    d->Rd = (w >> 0) & b111;
    d->Rn = (w >> 3) & b111;
    d->Rm = (w >> 6) & b111;
/* CONTROL FLOWS START */
if(context_cond)
{
setRegTaint(d->Rd,(getRegTaint(d->Rn)|
     getRegTaint(d->d->Rm)|taint_cond))
}
/* CONTROL FLOWS END */
else
/* EXPLICIT  FLOWS START */
{
setRegTaint(d->Rd,(getRegTaint(d->Rn)|
     getRegTaint(d->d->Rm)));
}
/* EXPLICIT  FLOWS END */
return  0;
```

Listing 5: Instrumenting T_THUMB_3REG instruction to propagate taint in control flows.

5.2 Context Switching Functions

When the context switches from Java to native, we store the taint in native runtime stack for tracking information flow at native level. The Context Switching Functions module instruments JNI-related functions that ensure switching between the two contexts.

These functions allow Java code to invoke native code. We hook the JNI call bridge (dvmCallJNIMethod) to detect native invoked methods. Then, we assign to each invoked native method a SourcePolicy structure where arguments taints of this method are stored. Finally, we add taints to the corresponding native context registers and memories.

Moreover, these functions allow native code to call Java methods through the dvmCallMethod function. We save taints in shadow registers and memory at native level and we use them to set taints in the Dalvik virtual machine stack when native codes invoke Java codes by instrumenting the dvmInterpret method. In addition, we maintain taint of a new Java object that can be created in the native codes through JNI functions. Furthermore, we instrument functions that allow native code to access to the Java objects' fields to assign taint to these object fields. Finally, we taint exceptions thrown by native code to communicate with Java code by instrumenting functions including "ThrowNew", "initException", "dvmCallMethod" and "dvmInterpret".

5.3 Information Kernel

The Information Kernel module provides Android Linux kernel information of processes and memory maps. We use the virtual machine introspection [16, 21, 24] technique described in Droidscope for reconstructing the OS-level view. The QEMU emulator disassembles and translates a basic block of guest instructions into an intermediate representation called TCG (Tiny Code Generator). Then, it compiles the TCG code block down to a block of host instructions and stores it in a code cache. To extract OS-level information (running processes, and the memory map) we instrument translated code blocks and we add TCG instructions at the code translation phase.

Table 3: Third party analysed applications.

Third party applications
The Weather Channel; Cestos; Solitaire; Babble; Manga Browser; Box2D*; Libgdx*; Knocking; Coupons; QQPhoneBook*; Fruit Ninja*; Bump; Traffic Jam; Find It*; Hearts; Blackjack; Alchemy; Horoscope; Bubble Burst Free; Wisdom Quotes Lite; Paper Toss*; ePhone*; Classic Simon Free; Astrid; Angry Birds*; Subway Surfer*; Layar; Cocos2D*; Unity*; Trapster; ProBasketBall; Grindr*; HeyWire*; Wertago; Dastelefonbuch*; RingTones; Yellow Pages; Contact Analyser; Hike*; TextPlus*

5.4 Native Taint Sink

The sensitive data can be leaked through native code. Thus, it is necessary to implement taint sink at native level. The Native Taint sink module hooks a selected system call to detect the leakage of private information. To make system calls, we use the service zero instruction *svc*#0. Thus, we instrument this instruction to get system call information. We hook file open ($fopen$), close ($fclose$), read ($fread$), write ($fwrite, fputc, fputs$) and connect ($send, sendto$) system calls to implement the native sinks. Our approach can be proved complete by following the same formal proof of completeness (soundness and compactness) described in (Graa et al., 2013).

6 EVALUATION

In this section, we evaluate effectiveness and performance of our approach. First, we analyse real popular Android applications using our system. Then, we evaluate our taint tracking approach overhead using CF-Bench. We use a Dell laptop (Latitude E6420) with a core i5 @ 2.6 GHz and 4GiB of RAM running Debian 7. Experiments were performed on an Android emulator version 4.1 that runs on our laptop. We enhance the approach implemented in (Graa et al., 2014) for tracking information flows (explicit and control flows) in native libraries.

6.1 Effectiveness

We download and analyse 40 free frequently used Android applications from the Google Play store (see Table 3). These applications access and handle private data such as location, contacts, phone state, camera and SMS. As shown in Table 3, 16 applications (marked with *) invoke native libraries.

We use dex2jar tool (Google, 2015) to translate dex files of different applications to jar files. Then, we use jd-gui (Java, 2015) to obtain the Java source code that will be analysed. For native code, we disassemble the libraries object code and we get the assembler mnemonics for the machine instructions by executing objdump (part of the GNU Binutils) (sourceware, 2015).

Table 4: Third party applications leaking sensitive data (L: location, Ca: camera, Co: contacts, P: phone state, SMS: messages) through information flows.

Application	Java context		Native Context		Type of leaked data
	Explicit flow	Control flow	Explicit flow	Control flow	
Wisdom Quotes Lite	X				L, P
The Weather Channel		X			L
Knocking		X			L, Ca, P
Coupons	X				L, Ca, P
QQPhoneBook			X		Co, SMS
Fruit Ninja				X	Co
Find It		X		X	L, P
Horoscope		X			L, P
Paper Toss		X		X	L, P
ePhone			X		Co
Astrid	X				L, P
Angry Birds				X	L
Subway Surfer				X	L
Layar	X				L, Ca, P
Trapster	X				L, Ca, P
Grindr				X	L, SMS
HeyWire				X	L, SMS
Wertago		X			L, Co, P
Dastelefonbuch		X		X	L, Co, P
RingTones	X				L, Co, P
Yellow Pages		X			L, Co, P
Hike				X	L, SMS
TextPlus				X	L, SMS

We found that 23 Android applications (see Table 4) leak private data through information flows:

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

- Six applications use explicit flow in Java context and cause the leakage of sensitive information like location, phone state, contact and camera.
- Eight applications use control flow in Java context and cause the leakage of sensitive information like location, phone state, contact and camera.
- Two applications use explicit flow in native context and cause the leakage of sensitive information like contact and SMS.
- Ten applications use control flow in native context and cause the leakage of sensitive information like location, phone state, contact and SMS. Most of these applications belong to Game category (Fruit Ninja, Angry Birds, Subway Surfer) and to communication category (Grindr; HeyWire, Hike, TextPlus). These ten applications are detected only by our approach.

TaintDroid detects 26% of malicious applications that cause leakage of sensitive data, NDroid and Droidscope 35%, Graa *et al.* implementation 61% and our approach detects all applications. So, our approach identifies more malicious applications that existing approaches in Android systems are unable to detect.

6.2 False Negatives

TaintDroid using its simple JNI tainting policy generates 74%, NDroid 65%, Droidscope 65% and Graa *et al.* approach 39% of false negatives. Our approach solves the under tainting problem. It has successfully propagated taint in control instructions at Java and native levels and detected leakage of tainted sensitive data that is reported in the alert messages.

6.3 Performance

To evaluate the performance of our approach, we study our static and dynamic analysis overhead. We perform the static analysis at load and verification time. Our approach adds 37%, and 30% overhead with respect to the unmodified system respectively at load and verification time. This, is due to the verification of method instructions and the construction of the control flow graphs used to detect the control flows.

We use CF-Bench (Bench, 2011) which is a CPU and memory benchmark tool to study our taint tracking approach overhead. We choose CF-Bench because it produces a fairly stable score, and tests both native as well managed code performance.

As shown in Figure 2, our approach incurs in average a slowdown of 14.9 times. This time of overhead is greater than the result of NDroid (5.45 times slowdown) because we propagate taint in the conditional branches at Java and native levels. This time of overhead is greater than the result of Droidscope (11 times slowdown) performed in a real machine and not in a Google Android Emulator that is prohibitively slow. Despite the proposed approach having additional speed loss, it gives more accurate detection results than NDroid and Droidscope.

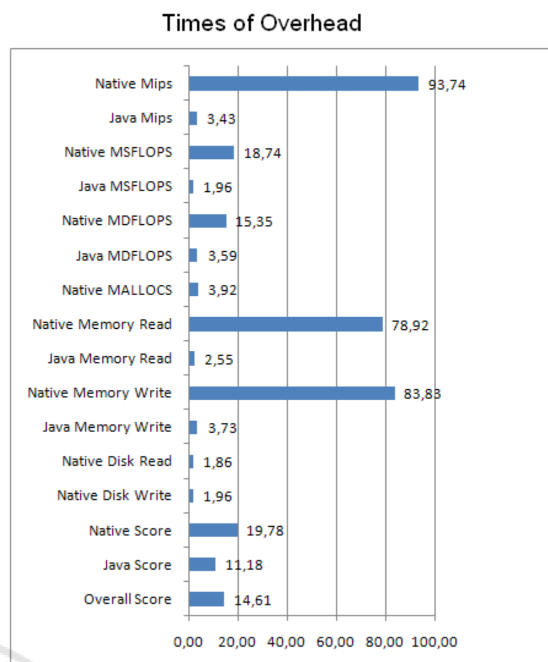

Figure 2: CF-Bench results of our taint tracking approach overhead.

6.4 False Positives

Our approach generates 30% of false positives. We detected leakage of IMSI and IMEI that was really used as a configuration parameter in the smartphone. So, we cannot consider these applications as malicious.

7 DISCUSSION

In this paper, we are interested in solving the under tainting problem because the false negatives can lead to a flaw in security. So, we taint all variables on conditional branches to reflect the control dependencies. Our approach generates false alarms that occured through incorrect interpretation of tainted data. To balance (trade-off) between over-tainting and leakage of private information, we can apply expert rules (ad hoc rules). These rules can be configured to specify what sensitive data are allowed to be sent without warning message. They depend on the sources of the sensitive data to be sent and their transmission contexts.

8 CONCLUSIONS

The untrusted applications installed on Android system can provoke the leakage of private data by exploiting control flows at native level. In this paper, we have proposed a hybrid approach that propagates taint along control dependencies in native librairies. We use static analysis to detect branches in the conditional structure and we complete with dynamic analysis to taint variables that depend on sensitive condition. Our approach is considered a complete work because it ensures tracking information flows (explicit and control flows) at Java and native levels in the Google Android operating system. It is a new approach for protecting private data from being leaked through control flows in native libraries. We evaluate our work through third party Android applications and we show that our approach detects effectively leakages of sensitive information by exploiting control flows in native librairies with reasonable performance overheads.

REFERENCES

Aho, A. V., Sethi, R., and Ullman, J. D. (1986). *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Allen, F. E. (1970). Control flow analysis. *ACM Sigplan Notices*, 5(7):1–19.

Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., and McDaniel, P. (2014). Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *ACM SIGPLAN Notices*, 49(6).

Bench (2011). Cf-bench. http://bench.chainfire.eu/.

Blasing, T., Batyuk, L., Schmidt, A.-D., Camtepe, S. A., and Albayrak, S. (2010). An android application sandbox system for suspicious software detection. In *Malicious and unwanted software (MALWARE), 2010 5th international conference on*, pages 55–62. IEEE.

Brown, J. and Knight Jr, T. (2001). A minimal trusted computing base for dynamically ensuring secure information flow. *Project Aries TM-015 (November 2001)*.

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

Cavallaro, L., Saxena, P., and Sekar, R. (2008). On the limits of information flow techniques for malware analysis and containment. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 143–163. Springer.

Chin, E., Felt, A. P., Greenwood, K., and Wagner, D. (2011). Analyzing inter-application communication in android. In *Proceedings of the 9th international conference on Mobile systems, applications, and services*, pages 239–252. ACM.

Clause, J., Li, W., and Orso, A. (2007). Dytan: a generic dynamic taint analysis framework. In *Proceedings of the 2007 international symposium on Software testing and analysis*, pages 196–206. ACM.

Denning, D. (1975). *Secure information flow in computer systems*. PhD thesis, Purdue University.

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

Enck, W., Gilbert, P., Chun, B., Cox, L., Jung, J., McDaniel, P., and Sheth, A. (2010). Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, pages 1–6. USENIX Association.

Enck, W., Octeau, D., McDaniel, P., and Chaudhuri, S. (2011). A study of android application security. In *USENIX security symposium*.

Fedler, R., Kulicke, M., and Schütte, J. (2013). Native code execution control for attack mitigation on android. In *Proceedings of the Third ACM workshop on Security and privacy in smartphones & mobile devices*, pages 15–20. ACM.

Fenton, J. (1974). Memoryless subsystem. *Computer Journal*, 17(2):143–147.

Fuchs, A. P., Chaudhuri, A., and Foster, J. S. (2009). Scandroid: Automated security certification of android applications. *Manuscript, Univ. of Maryland*.

Google (2015). dex2jar. http://code.google.com/p/dex2jar/.

Graa, M., Cuppens-Boulahia, N., Cuppens, F., and Cavalli, A. (2012). Detecting control flow in smarphones: combining static and dynamic analyses. In *Proceedings of the 4th international conference on Cyberspace Safety and Security*, pages 33–47, Berlin, Heidelberg. Springer-Verlag.

Graa, M., Cuppens-Boulahia, N., Cuppens, F., and Cavalli, A. (2013). Formal characterization of illegal control flow in android system. In *Proceedings of the 9th International Conference on Signal Image Technology & Internet Systems*. IEEE.

Graa, M., Cuppens-Boulahia, N., Cuppens, F., and Cavalli, A. (2014). Detection of illegal control flow in android system: Protecting private data used by smartphone apps. In *Foundations and Practice of Security*, pages 337–346. Springer.

Grace, M., Zhou, Y., Zhang, Q., Zou, S., and Jiang, X. (2012). Riskranker: scalable and accurate zero-day android malware detection. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*, pages 281–294. ACM.

Hornyack, P., Han, S., Jung, J., Schechter, S., and Wetherall, D. (2011). These aren't the droids you're looking for: retrofitting android to protect data from imperious applications. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 639–652. ACM.

Java (2015). Java decompiler. http://jd.benow.ca/.

Kang, M., McCamant, S., Poosankam, P., and Song, D. (2011). Dta++: Dynamic taint analysis with targeted control-flow propagation. In *Proc. of the 18th Annual Network and Distributed System Security Symp. San Diego, CA*.

Nair, S., Simpson, P., Crispo, B., and Tanenbaum, A. (2008). A virtual machine based information flow

control system for policy enforcement. *Electronic Notes in Theoretical Computer Science*, 197(1):3–16.

Portokalidis, G., Homburg, P., Anagnostakis, K., and Bos, H. (2010). Paranoid android: versatile protection for smartphones. In *Proceedings of the 26th Annual Computer Security Applications Conference*, pages 347–356. ACM.

Qian, C., Luo, X., Shao, Y., and Chan, A. T. (2014). On tracking information flows through jni in android applications. In *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on*, pages 180–191. IEEE.

Reina, A., Fattori, A., and Cavallaro, L. (2013). A system call-centric analysis and stimulation technique to automatically reconstruct android malware behaviors. *EuroSec, April*.

Rob van der Meulen, J. R. (2013). Gartner says smartphone sales accounted for 55 percent of overall mobile phone sales in third quarter of 2013. http://www.gartner.com/newsroom/id/2623415.

Song, D., Brumley, D., Yin, H., Caballero, J., Jager, I., Kang, M., Liang, Z., Newsome, J., Poosankam, P., and Saxena, P. (2008). Bitblaze: A new approach to computer security via binary analysis. *Information Systems Security*, pages 1–25.

sourceware (2015). Objdump. https://sourceware.org /binutils/docs/binutils/objdump.html.

Spreitzenbarth, M., Freiling, F., Echtler, F., Schreck, T., and Hoffmann, J. (2013). Mobile-sandbox: Having a deeper look into android applications. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, pages 1808–1815. ACM.

WANG, C. and SHIEH, S. W. (2015). Droit: Dynamic alternation of dual-level tainting for malware analysis. *JOURNAL OF INFORMATION SCIENCE AND ENGINEERING*, 31:111–129.

Warren, C. (2013). Google play hits 1 million apps. http://mashable.com/2013/07/24/google-play-1-million/.

Wiki (2015). Qemu open source processor emulator. http://wiki.qemu.org/Main_Page/.

Yan, L.-K. and Yin, H. (2012). Droidscope: Seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. In *USENIX Security Symposium*, pages 569–584.

Yin, H., Song, D., Egele, M., Kruegel, C., and Kirda, E. (2007). Panorama: capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, pages 116–127. ACM.

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

SCIENCE AND TECHNOLOGY PUBLICATIONS