2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks

On Tracking Information Flows through JNI in Android Applications

Chenxiong Qian[†], Xiapu Luo[†‡§], Yuru Shao[†], and Alvin T.S. Chan[†]
Department of Computing, The Hong Kong Polytechnic University[†]
The Hong Kong Polytechnic University Shenzhen Research Institute[‡]
{cscqiang,csxluo,csyshao,cstschan}@comp.polyu.edu.hk

***Abstract*—Android provides native development kit through JNI for developing high-performance applications (or simply apps). Although recent years have witnessed a considerable increase in the number of apps employing native libraries, only a few systems can examine them. However, none of them scrutinizes the interactions through JNI in them. In this paper, we conduct a systematic study on tracking information flows through JNI in apps. More precisely, we first perform a large-scale examination on apps using JNI and report interesting observations. Then, we identify scenarios where information flows *uncaught* by existing systems can result in information leakage. Based on these insights, we propose and implement `NDroid`, an efficient dynamic taint analysis system for checking information flows through JNI. The evaluation through real apps shows `NDroid` can effectively identify information leaks through JNI with low performance overheads.**

I. Introduction

The popularity of Android platform is evident from the tremendous number of activated devices and available applications. As of Sept. 2013, there are around one billion activations and 1M apps in the Google Play market [1]. Although most apps were developed in pure Java, Android's native development kit (NDK) offers developers enormous opportunities to extend apps using the Java native interface (JNI), such as employing OpenGL ES and OpenSL ES, for better performance, re-using native codes in C/C++, etc. Since Android 2.3, developers can even create an entire app using native codes. Recent years witnessed a considerable increase in the number of Android apps employing native libraries. For example, from 204,040 applications collected in May-Jun. 2011 from several markets, Zhou et al. identified 4.52% of them using native codes [2]. This percentage increased to 9.42% in 118,318 apps collected by the same authors in Sept.-Oct. 2011 [3]. We downloaded 227,911 apps from the Google Play market for a year (from Jun. 2012 to Jun. 2013) and found that 16.46% of them use native libraries. A recent study showed that 24% apps crawled from Asian third-party mobile markets contain native code [4]. At the same time, malware also uses NDK to hide the program logic and impede reverse engineering [2, 5].

Although there are many systems for analyzing apps or detecting malware[2, 3, 6], only a few of them inspect the native libraries in apps. However, none of them scrutinize the interactions between an app's Java codes and its native codes, which may lead to security loopholes.

The dynamic taint analysis could overcome this shortcoming because it inspects the information flow when the tainted data is propagated through the program [7, 8]. Unfortunately, existing dynamic taint analysis systems for Android, including `Taintdroid` [9] and `Droidscope` [10], are limited in the taint propagation logic related to JNI and its performance, because they were not designed specifically for apps using NDK. On one hand, although `Taintdroid` could achieve real-time information flow checking, we found that `Taintdroid` under-taints explicit information flows from native code to Dalvik virtual machine (DVM). On the other hand, `Droidscope`'s overhead is quite high, because it reconstructs OS level and DVM level information only from the machine instructions without exploiting JNI's semantic information. Moreover, its capability could be restricted by Just-In-Time compilation. Note that no new information flows than `Taintdroid` were reported in [10].

In this paper, we conduct a systematic study on tracking information flows through JNI in apps. We first perform a large-scale examination on apps using JNI, which are identified from a set of 227,911 apps crawled from the Google Play market. The number of examined apps is much larger than that in previous works [2–4]. We observe interesting behaviors on how apps utilize native libraries and report them in Section III. Then, we identify scenarios where information flows *uncaught* by existing dynamic taint analysis systems can result in information leakage. As a result, malicious apps can employ such information flows to leak sensitive data without being noticed by existing systems. This has motivated us to build a new system that can capture these information flows.

Based on these insights, we propose and implement `NDroid`, an efficient dynamic taint analysis system that tracks information flows cross the boundary between Java code and native code and the information flows within native codes. `NDroid` also works seamlessly with `TaintDroid` to track information flows from selected sources to specified sinks in apps. To make `NDroid` effective and efficient, we tackle many challenging issues, such as, multilevel function hooking, ARM/Thumb instruction instrumentation, etc. The evaluation through real apps with native libraries (e.g., QQPhoneBook v3.5, etc.), which can circumvent existing

§ The corresponding author.

978-1-4799-2233-8/14 $31.00 © 2014 IEEE
DOI 10.1109/DSN.2014.30

systems, demonstrates NDroid's effectiveness in discovering information leaks through JNI. We further evaluate NDroid's performance using public benchmark tool and find that NDroid introduces much lower overhead than [10].

The rest of this paper is organized as follows. Section II introduces the background and related work. Section III reports the study of 37,506 apps using native codes. Section IV describes the scenarios of information leaks through JNI. We detail the design, implementation, and evaluation of NDroid in Section V and Section VI. After discussing NDroid's limitations in Section VII, we conclude the paper in Section VIII.

II. Background

A. Java native interface and Android NDK

JNI facilitates the interoperation between Java and native libraries [11]. On one hand, using JNI, Java codes can pass parameters to native functions and obtain the return values after invocations. On the other hand, the JNI allows native codes to create and manipulate Java objects (e.g., invoking methods and accessing fields). To improve apps' performance, Android supports JNI and provides a set of native libraries, tools, and header files through its NDK [12].

We introduce an Android feature that brings challenges to the design of NDroid. Since version 4.0, Android uses indirect references in native code rather than direct pointers to reference objects. By doing so, when the garbage collector (GC) moves an object, it updates the indirect reference table with the object's new location. Consequently, native codes will hold valid object pointers every time GC moves objects around [13]. To track information flows through JNI, NDroid has to handle both indirect references and direct pointers as explained in Section V.

B. Taintdroid

TaintDroid is an information-flow tracking system for monitoring sensitive information in Android [9]. By modifying Android's application framework and DVM, TaintDroid attaches tags (i.e., taints) to sensitive data, propagates the taints when apps are running, and checks whether the taints will reach selected sinks. However, it under-taints information flows through JNI as illustrated in Section IV. NDroid not only overcomes these limitations but also can work seamlessly with TaintDroid to track information flows in apps. For the ease of explaining NDroid in Section V, we introduce some major data structures in TaintDroid.

Stack Structure As shown in Fig. 1, TaintDroid modifies DVM's stack structure to increase stack size for storing taint labels related to registers. For method invocation, TaintDroid first stores the taint labels interleaved with the parameters at the current stack frame's outs area. Then it allocates stack slots for callee's local variables and lets the

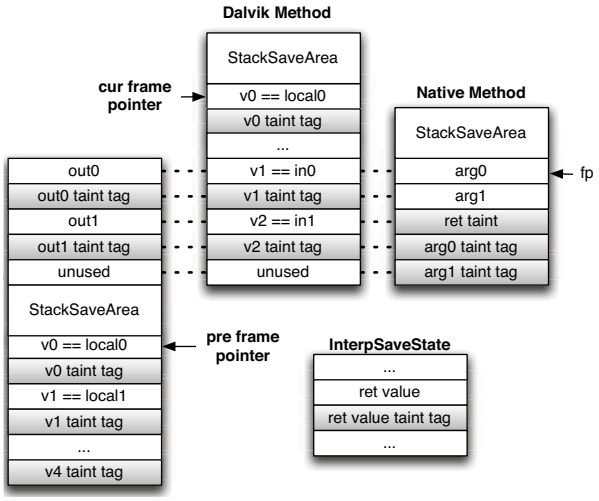


Figure 1. TaintDroid Stack Structure

frame pointer point to the new method's first local variable. After that, TaintDroid allocates a *StackSaveArea* on the top of the stack for saving the caller's information.

When a method returns, TaintDroid will save the return value's taint label into current thread's *InterpSaveState*. If the target is a native method, TaintDroid will store both the parameters' taint labels and the return value's taint label that is appended to the parameters. The return value's taint label is set by JNI Call Bridge according to TatintDroid's taint propagation policy, because native codes cannot directly access the return value's taint label. The retrun value's taint label will also be copied to current thread's *InterpSaveState* after the native method returns.

Taint Storage For ArrayObject and StringObject that is actually an array of chars, TaintDroid sets a taint label in the array object. For class static field and class instance field, the taint labels are stored interleaved with variables in Class's or Object's instance data area. For other Java objects, TaintDroid only keeps the taint label of their references.

Taint Propagation The taint propagation policy is a set of rules that define when and how taint should be propagated. TaintDroid adds taints to the sources of sensitive information (*GPS data, SMS messages, IMSI, IMEI, etc.*) of an Android device. The taint labels in TaintDroid are represented by 32bit integers, each bit of a taint label indicates one type of sensitive information, and different types of sensitive information are combined by the union operation of different taint labels. TaintDroid tracks the taints of primitive type variables and object references according to the logic of each DVM instruction. When a native method is called, TaintDroid adopts the taint propagation policy that the return value will be tainted if any parameter is tainted.

C. Related work

Only a few existing systems take into account the native libraries in Android applications. Some of them dynamically collect system calls through system call hijacking [14] or tools like `ptrace` [15], `strace` [16], and `ltrace` [4]. The sequence of system calls along with other function calls within DVM could then be used to characterize an application's behavior [17]. `CopperDroid` combines system calls obtained by instrumenting `QEMU` and Android specific behaviors observed from binder to detect malware [18]. Fedler et al. proposed measures to control the execution of native code on the Android platform [19]. Since dynamic analysis system is usually not scalable and could not cover all execution paths, static analysis approaches have been designed to scan native codes for detecting malware [3, 20]. However, static analysis is usually hindered by various obfuscation techniques [21].

Orthogonal to monitoring functions calls, information flow tracking empowers users to understand how a program processes tainted data [17]. There are two pioneering systems for this purpose: `TaintDroid` [9] and `DroidScope` [10]. `TaintDroid` modified DVM to carry out dynamic taint analysis and introduces low performance overhead. However, as illustrated in Section IV, it undertaints information flows through JNI. `AppFence` is based on `TaintDroid` and does not process third-party native libraries [22]. `DroidScope` tracks information flow at the instruction level by enhancing `QEMU` and it may incur 11 to 34 times slowdown [10]. Moreover, `DroidScope` did not report new information flows through JNI than `TaintDroid` [10, 23]. We identify the information flows missed by these systems and `NDroid` can capture them with much lower overhead than `DroidScope`.

The majority of existing security systems for Android do not consider native libraries. Instead, they usually inspect required permissions [2, 24], invoked APIs[2], and information flows within DVM [25]. The security of JNI in the Java virtual machine (JVM) has been investigated. Tan et al. discovered vulnerabilities in JNI based programs through static analysis [26] and designed sandbox to enable trustworthy execution of native codes [27]. `Jinn` defines 11 finite state machines and uses them to detect interface violations related to JNI [28]. Note that these sandboxes were designed for JVM instead of the DVM.

Dynamic taint analysis has been widely used in many applications, such as detecting vulnerabilities [29], malware analysis [30], understanding network protocols [31], to name a few [7, 8]. Despite many dynamic taint systems have been designed for either binary executables [7, 32, 33] or managed runtimes [34], there are still many open questions in dynamic taint analysis, such as conduct control flow taint and deal with implicit information flows [7, 8]. Although `NDroid` shares the limitations of dynamic taint analysis, it decreases the false negatives related to native codes by carefully tracking information flows through JNI.

III. Analysis of Apps Using JNI

From 227,911 apps fetched from the Google Play market, we pick out three types of apps that may use JNI for analysis, including (I) apps that invoke *System.load()* or *System.loadLibrary()* to load native libraries; (II) apps that contain native libraries without calling *System.load()* or *System.loadLibrary()*; (III) apps written in pure native code. Note that if the Java code in an app wants to invoke methods in native code, it has to first use either *System.load()* or *System.loadLibrary()* load the native library into the memory. Type I apps have explicitly called these methods. Although type II apps do not contain such invocations, as explained in the following paragraphs, we found that some apps may equip themselves with the capability to load native libraries by dynamically loading dex files containing the above invocations.

A. Type I apps

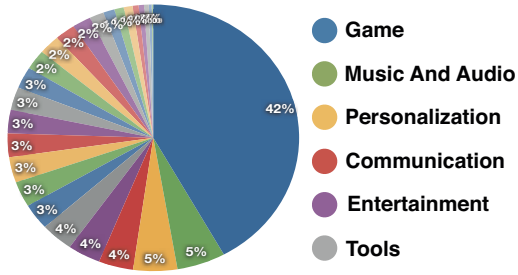


Figure 2. Native Libraries' Category Distribution

Category Distribution: There are 37,506 type I apps. Following the taxonomy of apps used by Google, we found that 42% of them belong to the *Game* category, as shown in Fig.2. It is as expected because game apps care their performance and many popular game engines are implemented in C/C++ code. The following game engines are widely used in the apps under investigation, include Unity, Box2D, Libgdx, and Cocos2D. Moreover, we found that apps in the category of "Music And Audio" always reuse existing native libraries and apps in the category of "Communication" often employ native code to hide communication protocols or encrypt data.

apps without libraries: 4,034 type I apps do *not* contain native libraries. We extracted the Java classes containing native method declarations from these apps and sorted these Java classes according to the number of applications using them. We identified eight classes, which belong to an *AdMob* plugin and are used by 48.1% of such apps. The dynamic analysis showed that they are repackaged apps with many advertisement components. Other reasons for such apps include (1) the required libraries have been loaded by the

system; (2) the App will not call the functions in native libraries but the related codes have not been deleted.

Library Distribution We collected the statistics of all the native libraries and manually analyzed 20 most popular libraries. Most of the libraries are from the famous game engine companies, such as Unity, Libgdx, Box2D, etc. There are a large portion of libraries relevant to video or audio processing. Other libraries, such as "libstlport_shared.so", "libcore.so", "libstagefright_froyo.so", etc, are originally included in NDK or the system. They are bundled with the applications for addressing Android's poor compatibility.

B. Type II apps

Among 1,738 type two apps, we found 394 apps that have the capability to load native libraries. More precisely, these apps have additional compressed dex files that can load native libraries. Therefore, once these apps dynamically load these dex files, they can load the native libraries. Note that many apps use similar approaches to hide the core business logic or enhance their functionality.

Other type two apps may not use their native libraries. One possible reason is that the native libraries would not be used during runtime (e.g., some libraries are for x86 and other platforms) but the developers forgot to remove them. For instance, for some libraries in open source projects, the codes for invoking them have been removed.

C. Type III apps

We only found 16 type three apps, including 11 game apps and 5 apps for entertainment. The small number of such apps may be due to the difficulty of developing such apps and the limitations of NDK APIs.

IV. Information leaks through JNI

In this section, we analyze the scenarios of leaking information through JNI, and explain why in some cases the information leaks cannot be detected by existing systems. Although currently there are, to the best of our knowledge, two dynamic taint analysis systems for Android (i.e., `Taintdroid` [9] and `Droidscope` [10]), we use `Taintdroid` as the representative because `Taintdroid` is open-source and available but the *taint tracker* in `Droidscope` has not been released yet. To detect information leaks, `Taintdroid` propagates the taint of sensitive source and checks whether it will reach any of the selected sinks in Java context. For native methods, `Taintdroid` taints the returned value of a JNI function if at least one parameter is tainted.

Information leakage occurs if there is an information flow from a sensitive source to a sink that can leak out the information. We regard the functions that can obtain sensitive information as the sources. The source and the sink can be in the Java context or the native context. If both the source and the sink are in the same context, the information flow through JNI must go through an intermediate in a *different* context. Table I lists the possible combinations of {source,intermediate,sink} in information flows through JNI. Since we do not consider the case when the source, the intermediate and the sink are in the same context, the corresponding cells are filled with "N/A". When both the source and the sink are in the Java context, there must be an intermediate in the native context as shown in case 1 and case 1'. Similarly, when both the source and the sink are in the native context, there is an intermediate in the Java context as shown in case 4. For case 2 and case 3, since the source and the sink are in different contexts, the intermediate's location does not matter to the analysis. As explained in the following paragraphs, `Taintdroid` can only detect case 1.

Table I
THE COMBINATIONS OF {SOURCE,INTERMEDIATE,SINK} IN INFORMATION FLOWS THROUGH JNI.

Sink		Java			Native	
Intermediate		Java	Native		Java	Native
Source	Java	N/A	Case 1	Case 1'	Case 2	
	Native	Case 3			Case 4	N/A

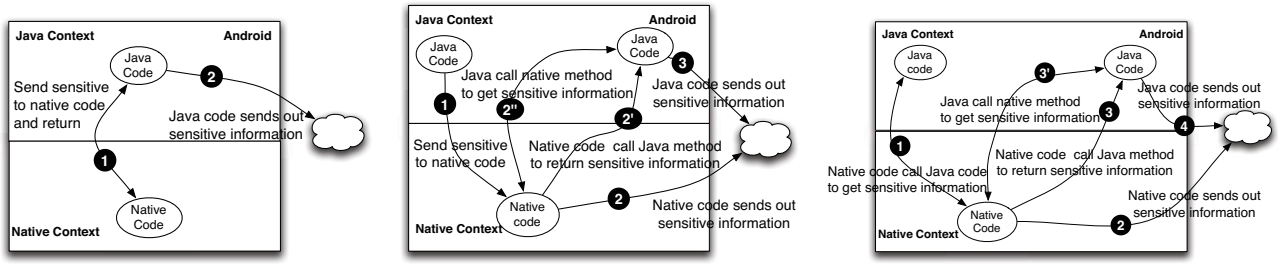
Case 1: After obtaining the sensitive data, the Java code calls native methods to process it and finally sends it to a sink. For example, as shown in Fig. 3(a), the Java code first calls a native method with parameters carrying sensitive data, collects the return value (i.e., step 1), and then sends it out (i.e., step 2). `Taintdroid` can detect such information leaks because it taints the method's return value.

Case 1': As shown in Fig. 3(b), the Java code invoking the native method with sensitive parameters will not send out the returned value (i.e., step 1). Instead, another piece of Java code fetches the sensitive information from the native method (i.e., step 2"), or the native code calls Java code to move the sensitive data from the native context to the Java context (i.e., step 2'). Finally, the Java code leaks the data (i.e., step 3). Since `Taintdroid` does not taint data obtained from a native method (e.g., data in step 2' and step 2"), it cannot detect such information leaks.

Case 2: As illustrated in Fig. 3(b), the native code will send the sensitive information out (i.e., step 2) after receiving it from the Java code (i.e., step 1). `Taintdroid` misses such leaks because it does not trace taint in the native context and its sinks do not include native methods.

Case 3: The native code collects sensitive data and passes it to the Java code for transmission. `Taintdroid` does not taint the data because it is collected by the native code. Fig. 3(c) illustrates that the native code can transmit the sensitive information obtained in step 1 to the Java context by calling the Java method (i.e., step 3) or waiting for the invocation from the Java code (i.e., step 3'). Finally, the Java code sends the information out (i.e., step 4).

Case 4: As shown in Fig. 3(c), the native code first gets



(a) Case 1 (b) Case 1' and 2 (c) Case 3 and 4

Figure 3. Examples of information leaks through JNI

the sensitive data from the Java context through JNI (i.e., step 1) and then leaks it (i.e., step 2). Similar to case 3, `Taintdroid` misses such leaks because it does not taint the data.

V. NDroid

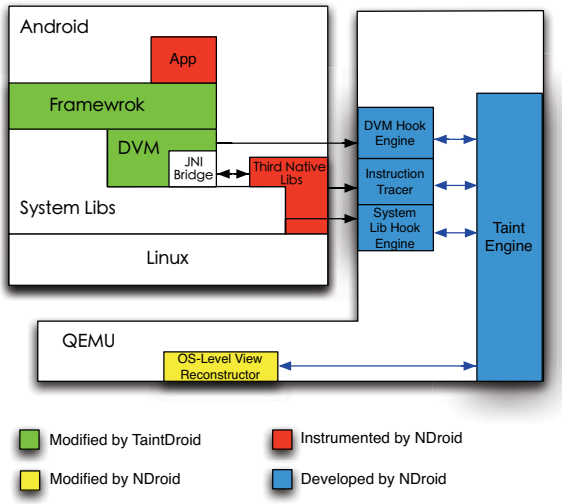


Figure 4. NDroid Architecture

A. Architecture

Android apps run in DVM on top of a modified Linux kernel with the support of Android application framework. The Android platform contains a set of system libs offering functions to the framework, DVM, and developers. Fig. 4 illustrates the architecture of `NDroid`, a virtualization-based dynamic taint analysis system. `QEMU` is an open-source machine emulator [35], through which we can get all ARM/Thumb instructions generated by the Android system. To track information flows through JNI, `NDroid` introduces four new modules into `QEMU` including (1) a DVM hook engine dealing with JNI related functions; (2) an instruction tracer processing ARM/Thumb instructions in native codes; (3) a system lib hook engine handling standard functions, and (4) a taint engine directing the taint propagation. We will detail them in the following subsections.

`NDroid` contains a customized OS-level view reconstructor motivated by `Droidscope` for obtaining the information of processes and memory map in Linux. Since `Taintdroid` carefully handles the taint propagation in the framework and DVM, we re-use the modules modified by `Taintdroid` and let the taints added by `NDroid` follow `Taintdroid`'s format so that they can work together smoothly.

B. DVM Hook Engine

A critical step in tracking information flow through JNI is to maintain and propagate taints between two different runtime contexts (i.e., the Java context and the native context). A challenging issue lies in how to correctly get and set taints when the context switches. For example, although `TaintDroid` properly handles the taints when an App is in the Java context, it does not store the corresponding taints to the native runtime stack when information flows enter the native context, thus failing to track such information flows. To address this issue, the DVM Hook Engine instruments important JNI-related functions, through which information flows cross the boundary between the Java context and the native context. These functions can be roughly classified into five groups according to their functionality, including (1) JNI entry; (2) JNI exit; (3) object creation; (4) field access; and (5) exception, each of which is detailed as follows.

JNI Entry: This category includes functions facilitating Java codes to invoke native methods. We define a structure `SourcePolicy` to record the taints to be propagated from the Java context to the native context. As shown in Listing 1, `SourcePolicy` includes *method_address*, the address of the native method's first instruction; *tR0 - tR3*, the taints of the first four parameters in registers R_0-R_3; *stack_args_num*, the number of remaining parameters on stack.

Note that the ARM/Thumb procedure call standard defines that the first four parameters are passed in R_0 to R_3, and the remaining parameters are pushed onto stack, and the return value is put in R_0 ; *method_shorty* describes the types of the parameters and the return value; *access_flag* indicates the method's access mode. Note that the first parameter of non-static method is "this"; *handler* points to the handler

responsible for completing the taint initialization, whose second parameter (i.e., *'CPUState'*) saves the runtime CPU state. Each native method receiving tainted parameters will have a `SourcePolicy` and we use a hash map to store the pairs of <addr, `SourcePolicy`>, where `addr` is the native method's address.

```
typedef struct _SourcePolicy{
  int method_address;
  int tR0, tR1, tR2, tR3;
  int stack_args_num;
  int* stack_args_taints;
  char* method_shorty;
  int access_flag;
  void (*handler) (struct _SourcePolicy*, CPUState*);
} SourcePolicy;
```

Listing 1. 'SourcePolicy'

```
void dvmCallJNIMethod(const u4* args, JValue* pResult,
     const Method* method, Thread* self);
```

Listing 2. 'dvmCallJNIMethod'

`NDroid` initializes the taint for tracking an information flow entering a native method in two steps. The first step creates and populates a `SourcePolicy` by hooking the method "dvmCallJNIMethod" (i.e., JNI Call Bridge), as showed in listing 2. More precisely, `NDroid` locates the parameters and their taints according to the first parameter of "dvmCallJNIMethod", which is the frame pointer. Note that these taints are set by the modified DVM. Moreover, we identify the *method_address*, *access_flag*, and *method_shorty* through the third parameter of "dvmCallJNIMethod", which points to the structure `Method`.

The second step adds taints to the native context. It occurs right before the native method executes. `NDroid` looks up the method's `SourcePolicy` from the hash map according to its address. Once found, based on the information on `SourcePolicy`, `NDroid` initializes the corresponding registers and memories with proper taint values.

Table II
JNI METHODS FOR INVOKING JAVA METHODS. **TYPE** ∈ {OBJECT, BOOLEAN, BYTE, CHAR, SHORT, INT, LONG, FLOAT, DOUBLE, VOID}

dvmCallMethodV	Call**Type**Method CallNonvirtual**Type**Method CallStatic**Type**Method
dvmCallMethodV	Call**Type**MethodV CallNonvirtual**Type**MethodV CallStatic**Type**MethodV
dvmCallMethodA	Call**Type**MethodA CallNonvirtual**Type**MethodA CallStatic**Type**MethodA

JNI Exit: This category includes functions helping native codes to call Java methods. The second column of Table II lists the methods used by native methods to call Java methods. These methods will eventually call the corresponding methods in the first column, which do similar things include (1) allocating the method frame on the DVM stack; (2) putting the parameters onto the stack; (3) scanning the parameters and converting the indirect reference of any object reference to the real object address through the method "dvmDecodeIndirectRef". We use "dvmCallMethod*" to denote these methods.

Note that neither the modified DVM nor Android's Linux kernel knows how to propagate taints associated with the parameters from the native context. `NDroid` accomplishes it by properly setting the taints in the DVM stack when native codes invoke Java methods through these functions.

It is challenging to handle these methods because of two reasons. First, the parameters of "dvmCallMethod*" do not contain the taint information. Second, when 'dvmCallMethod*' executes, it will clear the slots in the DVM stack, which are used to save the taints. To tackle the first issue, `NDroid` creates shadow registers and memory to save the taints in the native context and refers to them when the taints are propagated to the Java context.

To solve the second issue, `NDroid` hooks the "dvmCallMethod*" method and the "dvmInterpret" method that is called by "dvmCallMethod*". Instrumenting "dvmInterpret" is to set taints in the DVM stack. Hooking "dvmCallMethod*" is to get the indirect references of Java objects to be tainted. More precisely, in the native context, as the direct pointers of Java objects (i.e., the real address in memory) may be changed [13], the shadow memory uses the indirect reference as key to locate the taint information. Since the "dvmCallMethod*" method converts the indirect references to direct pointers and passes them to "dvmInterpret", we keep the indirect references for looking up the corresponding taint in the shadow memory.

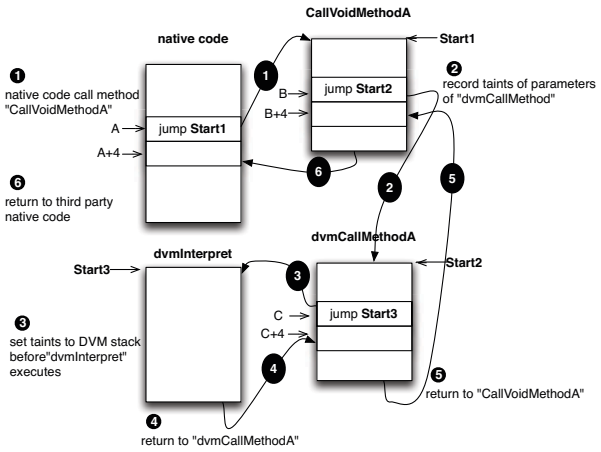


Figure 5. Multilevel Hooking

Since the methods "dvmCallMethod*" and "dvmInterpret" may also be invoked by other codes rather than the native codes under investigation, the overhead will be high if we hook these two functions whenever they are called.

Table III
JNI – CREATE NEW OBJECT

***Memory Allocation Function* (MAF)**	***New Object Function* (NOF)**
dvmAllocObject	NewObject, NewObjectV, NewObjectA
dvmCreateStringFromUnicode	NewString
dvmCreateStringFromCstr	NewStringUTF
dvmAllocArrayByClass	NewObjectArray
dvmAllocPrimitiveArray	New**PrimitiveType**Array

To address this issue, we propose a multilevel hooking technique to assure that the instrumentation of "dvmCallMethod*" and "dvmInterpret" is triggered only by the native codes under examination. Its basic idea is to define and check a sequence of preconditions before hooking certain methods.

We use the method "dvmCallVoidMethodA" as an example to explain the multilevel hooking technique, as shown in Fig. 5. We define six conditions T_1, T_2 ..., T_6 to determine whether the corresponding steps in Fig. 5 can be executed. Let $\boldsymbol{I}_{from}$ represent the address of the current instruction and $\boldsymbol{I}_{to}$ denote the target address of the jump instruction:

1) T_1 is true if $\boldsymbol{I}_{from}$ is within the native code and $\boldsymbol{I}_{to}$ equals the start address of "CallVoidMethodA".
2) T_2 is true if T_1 is true and $\boldsymbol{I}_{to}$ equals the start address of "dvmCallMethodA".
3) T_3 is true if T_2 is true and $\boldsymbol{I}_{to}$ equals the start address of "dvmInterpret".
4) T_4 is true if T_3 is true and $\boldsymbol{I}_{to}$ equals $C+4$, the address next to the instruction that calls "dvmInterpret".
5) T_5 is true if T_2 is true and $\boldsymbol{I}_{to}$ equals $B+4$, the address next to the instruction that calls "dvmCallMethodA".
6) T_6 is true if T_1 is true and $\boldsymbol{I}_{to}$ equals $A+4$, the address next to the instruction that calls "dvmCallVoidMethodA" in the native code.

With multilevel hooking, we can determine whether "dvmCallMethodA" (or "dvmInterpret") should be instrumented according to T_2 (or T_3).

Object Creation: Native codes can create new Java object through JNI functions listed in the second column of Table III, which are denoted as **NOF**. These functions will invoke the corresponding methods in the first column of Table III, which are denoted as **MAF**. **MAF** allocates memory for an object or an array. Note that **NOF** will convert the real object address returned by **MAF** to indirect reference. `NDroid` maintains the mapping between the indirect reference and the taint of the new object in the native context. The real object address is also required because `NDroid` needs to locate the newly created object (i.e., StringObject or ArrayObject) before tainting it. Therefore, to get the new object's indirect reference and real address, we apply the multilevel hooking technique to instrument both **NOF** and the corresponding **MAF**.

Field Access: Since native codes can access a Java object's fields through the functions listed in Table IV, by hooking these methods, `NDroid` can add taints to the corresponding field before executing *"Set*Field"* functions or get a field's taint after executing *"Get*Field"* functions.

Table IV
JNI METHODS TO GET/SET FIELD. **PRIMITIVE** $\in$ {BYTE, SHORT, INT, LONG, FLOAT, DOUBLE, BOOLEAN, CHAR}.

Get Field Functions	***Set Field Functions***
GetObjectField	SetObjectField
Get**Primitive**Field	Set**Primitive**Field
GetStaticObjectField	SetStaticObjectField
GetStatic**Primitive**Field	SetStatic**Primitive**Field

Exception: Native codes can communicate with Java codes through throwing an exception carrying sensitive information. The JNI function "ThrowNew" first creates a new exception object and then initializes it by invoking "initException", which creates a string object based on the third parameter of "ThrowNew" and calls the exception object's constructor through "dvmCallMethod". To track this information flow, we use the multilevel hooking technique to instrument functions including "ThrowNew", "initException", "dvmCallMethod" and "dvmInterpret", and add the taint of the third parameter of "ThrowNew" to the string object in the new exception object.

C. Instruction Tracer

By instrumenting third-party native libraries, the instruction tracer monitors each ARM/Thumb instruction to determine how the taint propagates. It takes time to decide each instruction because there are 148 ARM instructions and 73 Thumb instructions and each instruction does not have fixed bits to denote the opcode. To speed up the identification of the instruction type and the search of the handler, `NDroid` caches hot instructions and the corresponding handlers. Currently, `NDROID` only supports arithmetic and copy operations, while others will be included in our future work.

```
1  //void *memcpy(void *dest,const void *src,size_t)
2  void memcpy_handler(TrustCallPolicy* policy, CPUState* env
       , int isBegin){
3    if(isBegin){
4      int destAddr = env->regs[0];
5      int srcAddr = env->regs[1];
6      int nBytes = env->regs[2];
7      int i = 0;
8      for(; i < nBytes; i++){
9        //propagate the srcAddr's taint to destAddr
10       addTaint(destAddr + i, getTaint(srcAddr + i));}}}
```

Listing 3. 'memcpy' Taint Operation

Table V lists the taint propagation logic for ARM/Thumb instruction. We manually analyze all 148 ARM and 73 Thumb instructions and `NDROID` handles 101 ARM and 55 Thumb instructions that affect taint propagation. "binary-op" represents the binary operations(e.g., add, etc.); "unary-op" denotes the unary operation(e.g., NOT, etc.); "R_d",

Table V
TAINT PROPAGATION LOGIC FOR ARM/THUMB INSTRUCTIONS

Insn Format	Insn Semantics	Taint Propagation	Description
binary-op R_d, R_n , R_m	R_d = R_n op R_m	t(R_d) = t(R_n) OR t(R_m)	set R_d taint to R_n taint OR R_m taint
binary-op R_d, R_m	R_d = R_d op R_m	t(R_d) = t(R_d) OR t(R_m)	add R_m taint to R_d taint
binary-op R_d, R_m, #imm	R_d = R_m op #imm	t(R_d) = t(R_m)	set R_d taint to R_m taint
unary R_d, R_m	R_d = op R_m	t(R_d) = t(R_m)	set R_d taint to R_m taint
mov R_d, #imm	R_d = #imm	t(R_d) = TAINT_CLEAR	clear the R_d taint
mov R_d, R_m	R_d = R_m	t(R_d) = t(R_m)	set R_d taint to R_m taint
LDR* R_d, R_n, #imm	addr = Cal(R_n, #imm), R_d = M[addr]	t(R_d) = t(M[addr]) OR t(R_n)	set R_d taint to M[addr] taint OR R_n taint
LDM(POP) regList, R_n, #imm	startAddr = Cal(R_n, #imm), endAddr = Cal(R_n, #imm), {R_i, R_j} = {M[startAddr], M[endAddr]}	t({R_i, R_j}) = t(R_n) OR t({M[startAddr], M[endAddr]})	set R_i taint to M[startAddr] taint OR R_n taint, set R_{i+1} taint to M[startAddr+4] taint OR R_n taint, ..., set R_j taint to M[endAddr] taint OR R_n taint
STR* R_d, R_n, #imm	addr = Cal(R_n, #imm), M[addr] = R_d	t(M[addr]) = t(R_d)	set M[addr] taint to R_d taint
STM(PUSH) regList, R_n, #imm	startAddr = Cal(R_n, #imm), endAddr = Cal(R_n, #imm), {M[startAddr], M[endAddr]} = {R_i, R_j}	t({M[startAddr], M[endAddr]}) = t({R_i, R_j})	set M[startAddr] taint to R_i taint, set M[startAddr+4] taint to R_{i+1} taint, ..., set M[endAddr] taint to R_j taint

"R_n", and "R_m" indicate the ARM registers; "#imm" is the immediate number; "M[addr]" denotes the memory at address "addr"; "OR" represents the union operation; "Cal(R_n, #imm)" calculates the result based on "R_n" and "#imm"; "t(R_d)" represents the taint of register "R_d"; "t(M[addr])" denotes the taints of the memories starting from "addr"; "LDM"/"STM" denotes the load/store multiple values instruction and "POP"/"PUSH" represents the special case of "LDM"/"STM" where "R_n" = "SP". For "LDR" like instructions, we set the taint of "R_d" to the union of "t(M[addr])" and "t(R_n)", because "addr" is calculated based on "R_n" and "#imm". That is, if the tainted input is the address of an untainted value, the taint will be propagated to it.

Table VI
MODELED STANDARD METHODS

libc	memcpy, free, malloc, memset, strlen, strcmp, realloc, strcpy, memcmp, strncmp, memmove, sprintf, strncpy, fprintf, strchr, snprintf, calloc, strstr, atoi, strrchr, memchr, strcat, sscanf, vsnprintf, strcasecmp, strdup, strncasecmp, strtoul, sysconf, vsprintf, vfprintf, atol
libm	sin, pow, cos, sqrt, floor, log, strtod, strtol, exp, atan2, sinf, ceil, cosf, sqrtf, tan, acos, log10, atan, asin, ldexp, sinh, cosh, fmod, powf, atan2f, expf

D. System Lib Hook Engine

Since the system standard functions will be frequently called by native libraries, instrumenting every instruction in these standard functions will take a long time and incur heavy overhead. Instead, we model the taint propagation operations for popular functions listed in Table VI. They are selected after we analyzed 5,000 apps with native libraries. Using the function "memcpy" as an example, Listing 3 shows how to model its taint propagation operation.

Table VII
IMPORTANT STANDARD LIBRARY CALLS

fwrite*, fclose, fopen, fread, close, write*, fputc*, read, fputs*, open, fcntl, fstat, munmap, mmap, dlopen, stat, fgets, socket, connect, send*, recv, dlsym, bind, dlclose, ioctl, listen, mkdir, accept, select, getc, rename, sendto*, recvfrom, fdopen, mprotect, remove, kill, fork, execve, chown, ptrace, sysconf, Dalvik_dalvik_system_DexFile_openDexFile_bytearray

`NDroid` hooks selected system calls (e.g., file read/write, network, etc.) as listed in Table VII. Particulary, if the data carrying taint reaches calls with *, `NDroid` regards it as a possible information leak.

E. Taint Engine

`NDroid` maintains shadow registers to store the related registers' taints and a taint map to store the memories' taints. The taint granularity of `NDroid` is byte. The general propagation logic behind `NDroid` follows the "or" operation. That is, if `NDroid` propagates A's taint T_A to B, then B's taint T_B will be updated with "$T_B \cup T_A$". However, if the tainted operand is used as the memory address, `NDroid` will taint the memory at this address. Currently, the taint engine only handles arithmetic and move/load operations, while others will be included in future work.

F. OS-Level View Reconstructor

Motivated by `Droidscope`, `NDroid` employs virtual machine introspection to collect the information of processes and memory maps in Android's Linux kernel by only analyzing ARM/Thumb instructions [10].

G. Hooking functions through `QEMU`

`NDroid` realizes hooking functions by inserting TCG (Tiny Code Generator) instructions during `QEMU`'s code translation phase. More precisely, we insert TCG codes to

the beginning (and the end) of this function so that our analysis functions will be invoked before (and after) the execution of this function.

To hook the selected JNI functions and standard library calls, we manually disassemble "libdvm.so", "libc.so", "libm.so", etc. and determine the offsets of these functions. When examining an App, NDroid obtains the start addresses of the system libraries from the memory map through the OS-level view reconstructor. For both the selected JNI functions and standard library functions, NDroid maintains a list of their addresses and the corresponding analysis functions. When processing a branch instruction, if the target method is in the list, NDroid will call its analysis functions before/after the method is executed. The instruction tracer parses each ARM/Thumb instruction and calls the related handler to complete the taint propagation before the instruction is executed.

VI. Experiments

NDroid is implemented in QEMU with 20,261 lines of C/C++ code measured by CLOC 1.6 and 200 lines of Python scripts. Executing Taintdroid in the modified QEMU, NDroid employs it to run apps and track information flow in the Java context. NDroid handles the information flows through JNI.

It is worth noting that identifying all apps using JNI to leak information requires an input generation system that can exhaustively exercise those apps' functionality. Unfortunately, designing such a system is still an open problem and out-of-the-scope of this paper. In our experiment, we first used one simple tool(i.e., Monkeyrunner) to generate random input to drive those 37,506 apps using JNI. Since this tool may miss many functions involving JNI, we just found that QQPhoneBook3.5, a popular App that has 500,000-1,000,000 downloads in the Google Play market, may leak sensitive information through JNI. Then, we manually generated input and executed 8 randomly selected apps, which use JNI and are related to phone/SMS/contacts. NDroid found that 3 apps delivered the contact and SMS information to native code. One app (i.e., ephone3.3) further sends out the contact information through native code. Moreover, we use two proof-of-concept (PoC) apps (one for case 2 and the other one for case 3) to further evaluate NDroid's capability of tracking information leaks through JNI. Finally, following [10], we use the CF-Bench by Chainfire to evaluate NDroid's overhead.

Experiments were performed in a Virtual Box virtual machine with 1GB memory running Linux Mint (LDME MATE Edition) and the host is MacBook Pro (MD101xx/A) with a Core i5 @ 2.5GHz and 4GB of RAM. We run TaintDroid for Android 4.1 with 2.6.29 Linux kernel and XATTR support for the YAFFS2 filesystem in NDroid. We modified TaintDroid to enable it to load third-party native libraries.

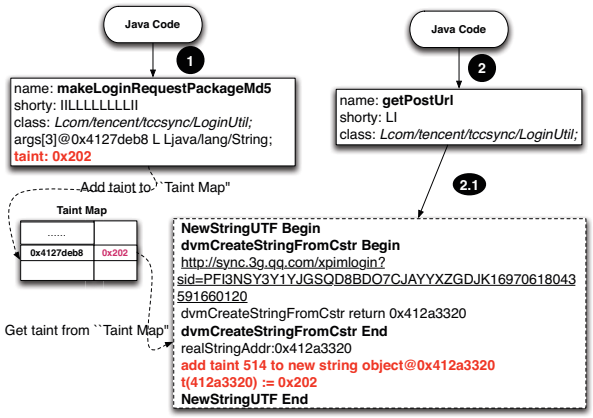


Figure 6. Log of QQPhoneBook

A. QQPhoneBook

NDroid found that QQPhoneBook3.5 may send sensitive information related to contacts and SMS to a server named "info.3g.qq.com". Fig. 6 shows the major functions in the information flow identified by NDroid, which is an example of Case 1'. In the first step, by invoking the native method "makeLoginRequestPackageMd5", the Java code transmits sensitive information through the fourth parameter (i.e., "args[3]") to the native context. This parameter is of the type String and its taint is "0x202". NDroid creates an entry in the taint Map to associate the memory address 0x4127deb8 with the taint "0x202".

Then the Java code calls another native method "getPostUrl" (i.e., step 2) with parameters that do not have taints. "getPostUrl" will invoke "NewStringUTF" (i.e., step 2.1) to create a new String object based on the tainted memory (i.e., 0x4127deb8) and return this new String object to the Java code that will eventually send out the sensitive data. NDroid not only adds a taint to the new String object and the return value but also tracks the information flow until it reaches the sink "send", thus capturing this information leakage. Note that TaintDroid alone cannot detect such information leakage because it does not taint the new String object and the return value of "getPostUrl".

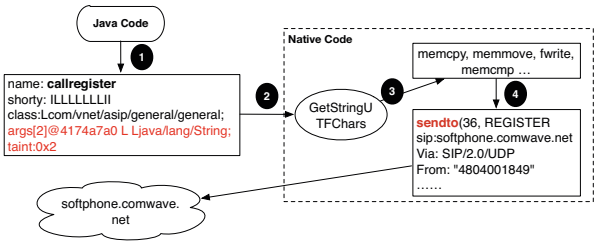


Figure 7. Log of ePhone

B. ePhone

NDroid found that ePhone3.3 may send contacts related information to a name named "soft phone.comwave.net".

Figure 8. PoC of case 2

Fig. 7 shows the major functions in the information flow tracked by NDroid. ePhone's Java code first calls a native method "callregister" that passes tainted information related to contacts to its native code. After that, the native code converts the tainted Java string to C string through the method "GetStringUTFChars" and further invokes many system calls, such as, "memcpy", "memmove", "fwrite", etc. to process the tainted information. Finally, it invokes "sendto" to send the tainted information to the server.

C. PoC of case 2 in information leakage

This PoC first fetches sensitive data by querying the contact information and then passes it to the native code that will write the data to a file. Fig. 8 depicts the major functions in this information flow.

By hooking "dvmCallJNIMethod", NDroid obtains the information of the invoked native method before its execution, such as the method's name (i.e., "recordContact"), class (i.e., "Lcom/ndroid/demos/Demos"), and the start address (i.e., 0x4a2c7d88). This method takes in three String parameters, all of which are tainted with the value "0x2", and returns a boolean value. NDroid constructs a SourcePolicy to record such information and save it into the hash map with the key value "0x4a2c7d88". When the native method's first instruction at "0x4a2c7d88" is executed, NDroid looks up the corresponding SourcePolicy and initializes the taints in the native context according to the information in SourcePolicy. More precisely, it sets the taint value "0x2" to memories at "0x5f80001d", "0x98000021" and "0xa9000025".

The native code converts Java strings to C strings through "GetStringUTFChars" (i.e., step 1, 2, 3) and obtains the contact id (i.e., "1"), contact name (i.e., "Vincent") and contact email (i.e., "cx@gg.com"). The taints are also propagated to memories at "0x2a141b90", "0x2a139060" and "0x2a1220d8". Then, the native code calls "fopen" (i.e., step 4) to open the file "/sdcard/CONTACTS", and the returned file pointer (i.e., **FILE***) is stored at "0x4006fd44". After that, "fprintf" is invoked to write the three stings to that file (i.e., step 5). Since "fprintf" is a sink, NDroid checks the parameters and notices that the three parameters are associated with the taint value "0x2". In step 6, the file is closed through "fclose".

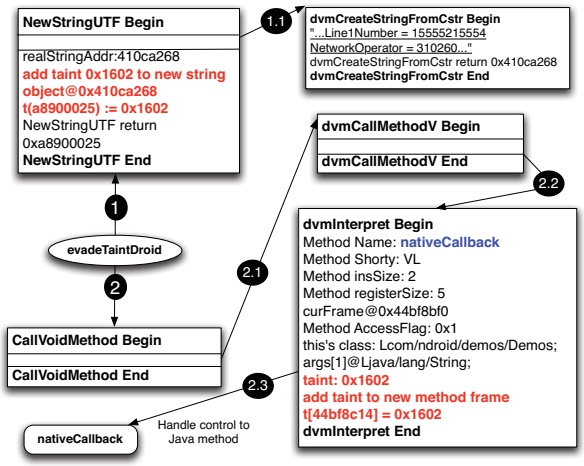


Figure 9. PoC of case 3

D. PoC of case 3 in information leakage

In this PoC, the Java code first obtains the device's information, including device ID, network Operator, etc. and then transfers it to the native context by calling the native method "evadeTaintDroid". After receiving the information, the native code creates a new Java String object to wrap the sensitive information by calling "NewStringUTF" (i.e., step 1) and then invokes the Java method "nativeCallback" (i.e., step2) to send out the information. Fig. 9 illustrates the major functions in this information flow.

By hooking "dvmCallJNIMethod", NDroid obtains the information of the native method "evadeTaintDroid" before its execution and sets the taints in the native context. The native method calls "NewStringUTF" (i.e., step 1) to create a new Java String object and gains the indirect reference "0xa8900025". "NewStringUTF" invokes "dvmCreateStringFromCstr" to create the Java String object and receives the real object address "0x410ca268" (i.e., step 1.1). By instrumenting "NewStringUTF", NDroid adds this method's parameter's taint value "0x1602" to the Java String object.

After that, the native code calls "CallVoidMethod" (i.e., step 2) which invokes "dvmCallMethodV" (i.e., step 2.1). Eventually, "dvmInterpret" is called (i.e., step 2.3) before the Java method "nativeCallback" executes. By instrumenting "dvmInterpret", NDroid obtains the Java method's information including method name ("nativeCallback"), method shorty ("VL"), method local variable size ("2"), method

register size (“5”), method’s frame address (“0x44bf8bf0”), and method access flag (“0x1”). Then, by checking each parameter’s taint and type, NDroid gets the first argument’s (i.e., “args[1]”) taint value (i.e., “0x1602”) and adds it to the Java method’s method frame slot at address “0x44bf8c14”. In step 2.3, the Java method “nativeCallback” is invoked to send out the tainted information. Since the network related methods are sinks, this information leakage is caught.

E. Performance

To measure NDroid’s performance, we ran CF-Bench 30 times on both NDroid and a vanilla QEMU with the Android platform. In average, NDroid incurs 5.45±0.414 times slowdown (showed in Fig. 10), which is much smaller than the result of Droidscope (i.e., at least 11 times slowdown). Note that our experiments were conducted in a virtual machine while the experiments in Droidscope were performed in a real machine with a similar configuration as our host of the virtual machine. The reason may be two-fold: (1) NDroid uses modified DVM and application framework to track information flows in the Java context whereas Droidscope does it through analyzing each ARM/Thumb instruction, which costs much time. (2) NDroid adopts several new approaches to increase its efficiency, such as, employing multilevel hooking to avoid unnecessary instrumentation, targeting on selected JNI functions, modelling the propagation logic of popular standard methods, and using caches to speed up the search, etc.

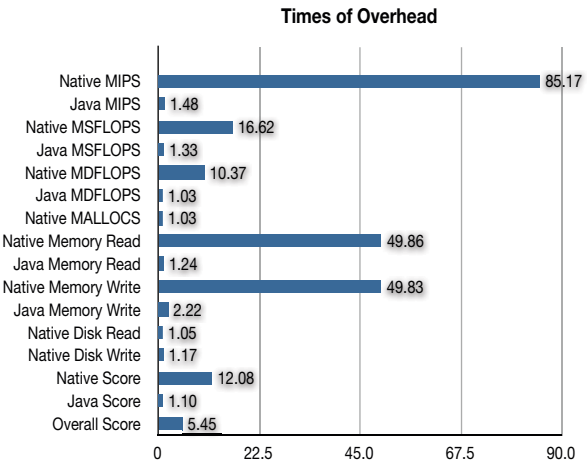


Figure 10. CF-Bench results

VII. Discussion

Similar to all dynamic analysis systems, NDroid executes one path at a time and cannot cover all execution paths. It is difficult to test apps because their behaviors are usually triggered by user interactions (e.g., clicking a button, turning off the screen) and they can extend their functionality through dynamical class loading. Experiment results in Section VI have showed that simple tools like monkeyrunner cannot enumerate all possible paths in an app and thus NDroid may miss information leakage. In future work, we will equip NDroid with advanced input generation system [36] to check apps.

We will realize a protection mechanism for taints before applying NDroid to analyze advanced malicious apps because they may modify or remove the taints. For example, an app without root privileges can manipulate the taints in DVM. With root privileges, an app can further manipulate stacks, modify trusted functions, and even establish the communication between Java and native code without following JNI specification. NDroid can be easily extended to protect taints and prevent evasions through stack manipulation or trusted function modification, because it monitors the memory, hooks major file and memory functions, and inspects every native instruction. Although we exclude apps with root privileges in this paper, NDroid can incorporate the functions in RootGuard [37], which monitors system calls for protecting rooted Android smartphones, to detect the abnormal behaviors of malware with root privileges.

Common to most virtualization-based systems is the difficulty of emulating the whole real hardware environment. The Android emulator misses some important information sources (e.g., GPS). Hence, NDroid cannot track information flows from these sources. On possible solution is to provide fake information that cannot be emulated as suggested by [38]. Moreover, advanced malware may exploit the difference between an emulator and a real smartphone to perform emulator detection. Using the virtualization technology supported by CPUs (e.g., Trustzone in ARM [39]) may be a promising approach to evade such detection.

Similar to TaintDroid and Droidscope, NDroid does not track control flows. Therefore, it could be evaded by apps that use the same control flow based techniques for circumventing those systems [40]. Since fully supporting control flow tracking may cause high overhead and false positives, we will investigate it and support more ARM/Thumb operations in future work.

VIII. Conclusion

We conduct a systematic study on tracking information flows through JNI in apps. Our large-scale examination on apps using JNI results in interesting observations on how apps use native libraries. We identify a set of scenarios where the information flows *uncaught* by existing systems can result in information leaks or characterize polymorphic malicious apps. Based on these insights, we propose and implement NDroid, an efficient dynamic taint analysis system for checking information flows through JNI, by tackling many challenge issues. The evaluation through real apps illustrates that NDroid can effectively identify information leaks through JNI and discover polymorphic malicious apps realized by JNI with low performance overheads. We will release NDroid later.

IX. Acknowledgment

We thank the reviewers for their comments and suggestions and Angelos Stavrou, in particular, for shepherding our paper. This work is supported in part by the Hong Kong ITF (No. ITS/073/12), the Hong Kong GRF (No. PolyU 5389/13E), the National Natural Science Foundation of China (No. 61202396), the Open Fund of Key Lab of Digital Signal and Image Processing of Guangdong Province, and Shenzhen City Special Fund for Strategic Emerging Industries (No. JCYJ20120830153030584)

References

[1] C. Smith, "25 amazing android statistics," http://expandedramblings.com/index.php/android-statistics/, Apr. 2014.

[2] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang, "Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets," in *Proc. NDSS*, 2012.

[3] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang, "Riskranker: Scalable and accurate zero-day android malware detection," in *Proc. MobiSys*, 2012.

[4] M. Spreitzenbarth, F. Echtler, and J. Hoffmann, "Mobile-sandbox: Having a deeper look into android applications," in *Proc. SAC*, 2013.

[5] Y. Zhou and X. Jiang, "Dissecting android malware: Characterization and evolution," in *Proc. IEEE Symp. Security and Provacy*, 2012.

[6] X. Wei, L. Gomez, I. Neamtiu, and M. Faloutsos, "Profiledroid: Multi-layer profiling of android applications," in *Proc. MobiCom*, 2012.

[7] E. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in *Proc. IEEE Secur. Pri. Symp.*, 2010.

[8] B. Livshits, "Dynamic taint tracking in managed runtimes," Microsoft Research, Tech. Rep. MSR-TR-2012-114, 2012.

[9] W. Enck, P. Gilbert, B. Chun, L. Cox, J. Jung, P. McDaniel, and A. Sheth, "Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones," in *Proc. USENIX OSDI*, 2010.

[10] L. Yan and H. Yin, "Droidscope: Seamlessly reconstructing OS and Dalvik semantic views for dynamic Android malware analysis," in *Proc. USENIX Sec*, 2012.

[11] S. Liang, *The Java Native Interface: Programmer's Guide and Specification*. Addison-Wesley, 1999.

[12] "Android NDK," http://developer.android.com/tools/sdk/ndk/index.html, 2013.

[13] E. Hughes, "JNI local reference changes in ICS," http://android-developers.blogspot.hk/2011/11/jni-local-reference-changes-in-ics.html, 2011.

[14] T. Blasing, L. Batyuk, A. Schmidt, S. Camtepe, and S. Albayrak, "An android application sandbox system for suspicious software detection," in *Proc. MALWARE*, 2010.

[15] G. Portokalidis, P. Homburg, K. Anagnostakis, and H. Bos, "Paranoid android: Versatile protection for smartphones," in *Proc. ACSAC*, 2010.

[16] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani, "Crowdroid: behavior-based malware detection system for android," in *Proc. SPSM*, 2011.

[17] M. Egele, T. Scholte, E. Kirda, and C. Kruegel, "A survey on automated dynamic malware-analysis techniques and tools," *ACM Computing Surveys*, vol. 44, no. 2, 2012.

[18] A. Reina, A. Fattori, and L. Cavallaro, "A system call-centric analysis and stimulation technique to automatically reconstruct android malware behaviors," in *Proc. EuroSec*, 2013.

[19] R. Fedler, M. Kulicke, and J. Schutte, "Native code execution control for attack mitigation on Android," in *Proc. SPSM*, 2013.

[20] A. Schmidt, R. Bye, H. Schmidt, J. Clausen, O. Kiraz, K. Yuksel, S. Camtepe, and S. Albayrak, "Static analysis of executables for collaborative malware detection on android," in *Proc. ICC*, 2009.

[21] A. Moser, C. Kruegel, and E. Kirda, "Limits of static analysis for malware detection," in *Proc. ACSAC*, 2007.

[22] P. Hornyaick, S. Han, J. Jung, S. Schechter, and D. Wetherall, "These arent the droids youre looking for: Retrofitting android to protect data from imperious applications," in *Proc. CCS*, 2011.

[23] L. Yan and H. Yin, "Presentation of DroidScope," https://www.usenix.org/conference/usenixsecurity12/droidscope-seamlessly-reconstructing-os-and-dalvik-semantic-views, Aug. 2012.

[24] H. Peng, C. Gates, B. Sarma, N. Li, Y. Qi, R. Potharaju, C. Nita-Rotaru, and I. Molloy, "Using probabilistic generative models for ranking risks of android apps," in *Proc. CCS*, 2012.

[25] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, "Chex: statically vetting android apps for component hijacking vulnerabilities," in *Proc. CCS*, 2012.

[26] G. Tan and J. Croft, "An empirical security study of the native code in the jdk," in *Proc. USENIX Sec*, 2008.

[27] M. Sun and G. Tan, "Jvm-portable sandboxing of java's native libraries," in *Proc. ESORICS*, 2012.

[28] B. Lee, B. Wiedermann, M. Hirzel, R. Grimm, and K. S. McKinley, "Jinn: Synthesizing dynamic bug detectors for foreign language interfaces," in *Proc. PLDI*, 2010.

[29] J. Newsome and D. Song, "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software," in *Proc. NDSS*, 2005.

[30] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda, "Panorama: Capturing system-wide information flow for malware detection and analysis," in *Proc. CCS*, 2007.

[31] G. Wondracek, P. Comparetti, C. Kruegel, and E. Kirda, "Automatic network protocol analysis," in *Proc. NDSS*, 2008.

[32] D. Zhu, J. Jung, D. Song, T. Kohno, and D. Wetherall, "Tainteraser: Protecting sensitive data leaks using application-level taint tracking," *SIGOPS Oper. Syst. Rev.*, vol. 45, no. 1, 2011.

[33] V. Kemerlis, G. Portokalidis, K. Jee, and A. Keromytis, "libdft: Practical dynamic data flow tracking for commodity systems," in *Proc. VEE*, 2012.

[34] V. Haldar, D. Chandra, and M. Franz, "Dynamic taint propagation for Java," in *Proc. ACSAC*, 2005.

[35] "Qemu," http://wiki.qemu.org/Main_Page, 2013.

[36] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: An input generation system for Android Apps," in *Proc. FSE*, 2013.

[37] Y. Shao, X. Luo, and C. Qian, "Rootguard: Protecting rooted android phones," *IEEE Computer*, June 2014.

[38] "Appuse - Android pentest platform unified standalone environment," https://appsec-labs.com/AppUse, 2013.

[39] ARM Ltd., "Trustzone," http://www.arm.com/products/processors/technologies/trustzone/index.php, visited 2014.

[40] G. Sarwar, O. Mehani, R. Boreli, and M. Kaafar, "On the effectiveness of dynamic taint analysis for protecting against private information leaks on Android-based devices," in *Proc. SECRYPT*, 2013.