NDroid: Toward Tracking Information Flows Across Multiple Android Contexts

Lei Xue, Chenxiong Qian, Hao Zhou, Xiapu Luo, Yajin Zhou, Yuru Shao, and Alvin T. S. Chan

***Abstract*— For performance and compatibility reasons, developers tend to use native code in their applications (or simply apps). This makes a bidirectional data flow through multiple contexts, i.e., the Java context and the native context, in Android apps. Unfortunately, this interaction brings serious challenges to existing dynamic analysis systems, which fail to capture the data flow across different contexts. In this paper, we first performed a large-scale study on apps using native code and reported some observations. Then, we identified several scenarios where data flow cannot be tracked by existing systems, leading to *uncaught* information leakage. Based on these insights, we designed and implemented `NDroid`, an efficient dynamic taint analysis system that could track the data flow between both Java context and native context. The evaluation of real apps demonstrated the effectiveness of `NDroid` in identifying information leakage with reasonable performance overhead.**

***Index Terms*— Android application analysis, taint analysis, Java native interface (JNI).**

I. Introduction

THE popularity of Android platform is evident from the tremendous number of activated devices and available apps. As of May 2017, there are around 72.68% smartphone running Android system [1]. At the same time, for better performance reason and compatibility of legacy code, developers tend to use native code in their apps and interface with Java code through the JNI bridge. Developers can even create an entire app using native code since Android 2.3.

Recent years witnessed a considerable increase in the number of apps using native libraries. For example, from 204,040 applications collected in May.-Jun. 2011 from several markets, Zhou *et al.* observed that 4.52% of them used native code [2]. This percentage increased to 9.42% in 118,318 apps collected by the same authors in Sep.-Oct. 2011 [3]. This trend is further confirmed by the findings that 24% apps crawled from Asian third-party mobile markets contain native code [4].

However, the popularity of native code in apps brings serious challenges to existing dynamic analysis systems. First, although there are many systems for analyzing apps or detecting malware [2], [3], [5], only a few of them inspect the native libraries in apps [6], [7], and none of them scrutinizes the interactions between an app's Java code and native code. This leads to a security loophole, which could be abused by malware to evade detection. Second, existing dynamic taint analysis systems, including TaintDroid [8] and TaintART [9], are limited in the taint propagation logic related to JNI. That is, these systems could under-taint explicit data flow from native code to Dalvik virtual machine (DVM), which we will show in Section III-C. Third, DroidScope [10] is able to track the whole system's data flow by design. However, the overhead is quite high since it reconstructs high level information from the underlying machine instructions without considering JNI's semantic information. This may limit the capability to track real data flow in Android apps in practice. For instance, compared with Taintdroid, no new data flows were reported by DroidScope.

In this paper, to fully understand the behaviors of apps using native libraries, we first performed a study on 319,725 apps crawled from Google Play and reported the results in Section III. Then we identified several scenarios where data flow cannot be tracked by existing dynamic taint analysis systems, which leads to *uncaught* information leakage. As a result, malicious apps can abuse these limitations to leak sensitive data and evade the detection. This motivated us to build a new system that can capture such information flows.

Based on these insights, we then designed and implemented `NDroid`, an efficient dynamic taint analysis system that could track data flow within native code, and more importantly cross the boundary between Java code and native code. We tackled multiple design and implementation challenges, including the support of different Android runtime, i.e., both DVM and ART runtime, multilevel function hooking, ARM/Thumb/Thumb2 instruction instrumentation, etc. `NDroid` can be used to detect information leakages through JNI and advanced malware samples that dynamically modifies its Dex file through native code [11], which cannot be detected by existing systems. The evaluation of real apps, which circumvent existing systems, demonstrated the effectiveness of `NDroid`. We further evaluated the performance

Manuscript received January 18, 2018; revised June 17, 2018; accepted July 27, 2018. Date of publication August 21, 2018; date of current version September 13, 2018. This work was supported in part by the Hong Kong GRF under Grant PolyU 152279/16E and Grant PolyU 152223/17E and in part by the Hong Kong RGC Project under Grant CityU C1008-16G. The associate editor coordinating the review of this manuscript and approving it for publication was Prof. Loukas Lazos. *(Lei Xue and Chenxiong Qian contributed equally to this work.) (Corresponding author: Xiapu Luo.)*

L. Xue, H. Zhou, and X. Luo are with the Department of Computing, The Hong Kong Polytechnic University, Hong Kong (e-mail: csxluo@comp.polyu.edu.hk).

C. Qian was with the Department of Computing, The Hong Kong Polytechnic University, Hong Kong. He is now with the School of Computer Science, Georgia Institute of Technology, Atlanta, GA 30332 USA.

Y. Zhou is with the Institute of Cyber Security Research, Zhejiang University, Hangzhou 310027, China, and also with the College of Computer Science and Technology, Zhejiang University, Hangzhou 310027, China.

Y. Shao was with the Department of Computing, The Hong Kong Polytechnic University, Hong Kong. He is now with the University of Michigan, Ann Arbor, MI 48109 USA.

A. T. S. Chan is with the Singapore Institute of Technology, Singapore 138683.

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TIFS.2018.2866347

overhead of NDroid and found that it introduced much lower overhead than DroidScope. In summary, our major contributions include:

- We analyzed multiple scenarios where information leakage could occur in Android apps, and shed light on the reasons why the data leakage cannot be detected by existing tools.
- We designed and implemented NDroid: a tool for analyzing apps with native code and can trace information leakage through multiple contexts (i.e., Java context and native context), which cannot be detected by existing systems. The new NDroid will be released after paper publication.
- We carefully evaluated NDroid and the results illustrated the effectiveness and efficiency of NDroid.

The rest of this paper is organized as follows. Section II introduces the background, and then we report the study of native libraries in Android framework APIs and 70,252 apps using native code in Section III. The high-level design of NDroid is introduced in Section IV, followed by the implementation on the DVM and ART in Section V and Section VI respectively. After introducing related works in Section VIII, we discuss our system's limitations in Section IX, and conclude our work in Section X.

II. Background

In this section, we will briefly introduce the necessary background information to better understand this paper.

A. Java Native Interface and Android NDK

Java native interface (JNI) facilitates the interaction between Java code and native libraries [12]. On one hand, using JNI, Java code can pass parameters to native functions and obtain the return values afterward. On the other hand, native code could create and manipulate Java objects (e.g., invoking methods and accessing fields) through JNI. To facilitate the usage of native libraries in apps, Android provides a set of libraries, tools, and header files through the NDK [13].

We find that there is a feature introduced by Android bringing challenges to our system. 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 code will hold valid object pointers every time GC moves objects around [14]. To track information flows through JNI, NDroid has to handle both indirect references and direct pointers.

B. The Off-the-Shelf Taint Propagation Tools

There are some existing systems that could track data flow in Android apps. In the following, we describe these off-the-shelf taint tracking systems.

1) TaintDroid: TaintDroid is an information-flow tracking system for monitoring sensitive information in Android [8]. 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

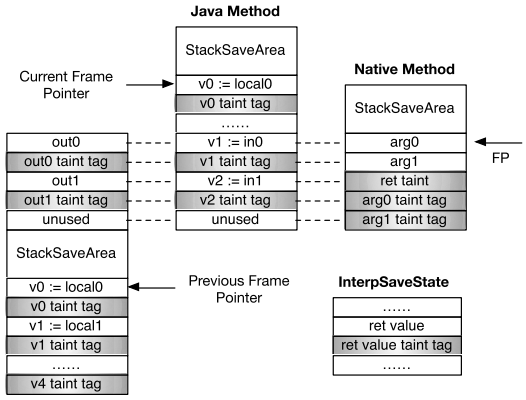


Fig. 1. TaintDroid stack structure.

information flows through JNI as explained in Section III. NDroid not only overcomes these limitations but also works seamlessly with TaintDroid on DVM to track information flows in apps. For the ease of explaining NDroid for DVM in Section V, we introduce the major data structures in TaintDroid.

a) 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 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 code 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.

b) Taint storage: For ArrayObject and StringObject that contain 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.

c) 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.

2) DroidScope: DroidScope [10] is an Android analysis platform based on the QEMU emulator [15]. It instruments machine instructions by adding extra TCG (Tiny Code Generator) instructions during the code translation phase. When such extra analysis code are executed, DroidScope can reconstruct OS-level knowledge, including processes, threads, system calls, and memory maps, and DVM-level knowledge, such as Dalvik instructions, DVM state, and Java objects. `NDroid` adopts DroidScope's OS-level view reconstructor, and extends it to do instrumentation at different level for different code (third party native libraries, system libraries, etc.). With flexible instrumentation on native code, `NDroid` completes taint propagation of native code at runtime. Besides, DroidScope does not support the new ART runtime.

3) TaintART and ARTist: Both TaintART [9] and ARTist [16] are proposed to conduct taint analysis on the apps running on ART. They propagate taint tags by inserting taint propagation instructions into the apps. In particular, they modify the Android system tool `dex2oat` to inject the taint propagation instructions when the apps' Dalvik code are compiled into native code by `dex2oat`. Unfortunately, they cannot insert such instructions into the Dalvik code that are not compiled into native code.

4) Malton: Malton [17] is proposed for only ART platform, and it is implemented based on the dynamic instrumentation framework Valgrind [18], which translates native instructions into IR statements during execution. Hence, Malton conducts taint analysis through inserting taint propagation IR statements dynamically, and then it propagates taint information following the logics of the executed IR statements.

III. Native Code on Android Platform

In this section, we first survey the usage of native libraries in Android platform and Android apps. Then, we analyze different scenarios of information leakage in Android apps, and explain why existing systems cannot detect such leakages.

A. Android Framework APIs

Android framework APIs provide apps the ability to conduct critical operations and communicate with the underlying Linux system. To fully understand how Android framework APIs are using native libraries, we customize PScout [19] to analyze 7 versions of Android framework spanning version 2.2.3 up to the recent released Android 7.0. On one hand, PScout provides us the statistics of the Android framework such as the number of public APIs (Num_{Public}) and native methods (Num_{Native}). On the other hand, we use the global call graph constructed by PScout to calculate the total number of public APIs that may reach native methods in their invocation chains (Num_{PubNat}). Note that PScout only treats methods within packages, such as "android.∗" and "com.∗", as the entry point of the invocation chain. We remove this constraint so that all public APIs become the entry points. Although Android framework comprises a relatively small number of native methods, more than 71% of public APIs invoke native methods in their implementations, as shown in Table I.

TABLE I

The Statistics of Framework APIs and the Percentage of Public APIs Which Call Native Methods in Seven Versions of Android

Version	Num_{Native}	Num_{Public}	Num_{PubNat}	$Percentage$
2.2.3	2590	73482	52720	71.75%
4.0.1	3113	97184	70218	72.25%
4.1.1	3151	107299	78041	72.73%
4.4	3646	120964	86261	71.31%
5.0	3972	145256	103764	71.44%
6.0	3657	164286	117764	71.68%
7.0	3864	162849	116736	71.68%

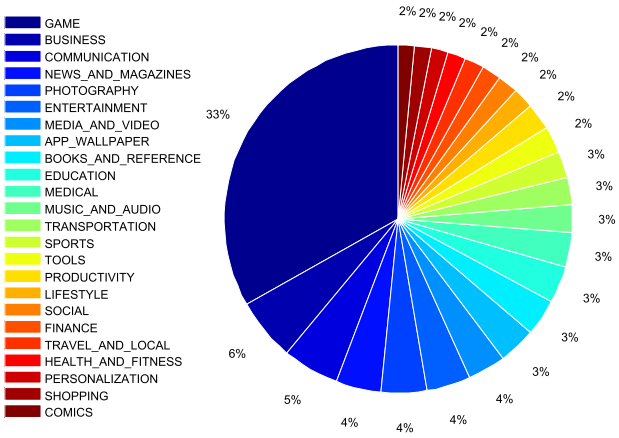


Fig. 2. The category distribution of native library.

B. Android Applications

We further analyze 319,725 apps downloaded from the Google Play. 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()* to load native libraries 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.

1) Type I Apps: There are 70,252 type I apps. Following the taxonomy of apps used by Google, we found that 33% 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, including 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.

In type I, 9,890 apps do *not* contain native libraries. We extracted the Java classes containing native method declarations from these apps and sorted these Java classes according

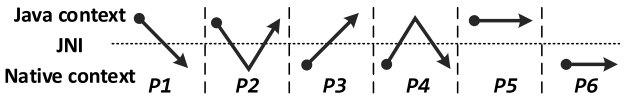


Fig. 3. The potential paths of information leakage.

to the number of applications using them. Consequently, 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 code have not been deleted.

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.

2) Type II Apps: Among 2,649 type II 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 the 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.

The other type II 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 code for invoking them have been removed.

3) Type III Apps: We only found 16 type III 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.

C. Information Leakage in Android Apps

In the following, we will analyze the basic scenarios of information leakage in Android apps, and then explain why some scenarios of the information leakages cannot be detected by existing systems.

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 are located in the Java context or the native context. Fig. 3 shows the six basic scenarios of information leakages in Android system. Since an app can combine several basic scenarios to create complicated scenarios, if the detection system misses one of the basic scenarios, it may also miss the whole complicated scenarios.

- *P1:* The Java code obtains the sensitive data and then passes it to the native code, which first processes the sensitive data and eventually leaks it out.
- *P2:* The sensitive data is collected by the Java code, and then passed to the native code for processing. After that, the result is obtained by the Java code again, and leaked in the Java context.
- *P3:* The native code gets the sensitive data and then passes it to the Java code, which first processes the sensitive data and eventually leaks it out.
- *P4:* The native code first acquires the sensitive data and then passes it to the Java code for processing. After that, the sensitive data is transmitted back to the native code, and it is eventually leaked in the native context.
- *P5:* Both the source and the sink are in the Java contest, and the sensitive data is processed only by the Java code.
- *P6:* Both the source and sink are in the native contest, and the sensitive data is processed only by the native code.

TABLE II
VARIOUS TAINT ANALYSIS TOOLS ON DVM|ART PLATFORM

Tools	*P1*	*P2*	*P3*	*P4*	*P5*	*P6*
`TaintDroid` [8]	○|○	◍|○	○|○	○|○	●|○	○|○
`Droidbox` [10]	●|○	○|○	○|○	○|○	●|○	●|○
`Androidperf` [20]	●|○	●|○	●|○	●|○	●|○	●|○
`ARTist` [16]	○|○	○|○	○|○	○|○	○|◍	○|○
`TaintART` [9]	○|○	○|○	○|○	○|○	○|◍	○|○
`TaintMan` [21]	●|○	○|○	○|○	○|○	◍|◍	○|○
`Malton` [17]	○|●	○|●	○|●	○|●	○|●	○|●
`NDroid`	●|●	●|●	●|●	●|●	●|●	●|●

Note: ● and ○ illustrate that the tool has or does not have such capacity respectively, whereas ◍ means the tool just supports special cases.

We also summarize the major capabilities of the popular taint analysis tools for Android in Table II. TaintDroid [8] and DroidScope [10] only support DVM. TaintDroid cannot support the information leakages of *P1*, *P3*, *P4* and *P6*, because it only handles the information leakages of which both the source and sink are in the Java context. The information leakage *P2* could be implemented by two ways. In the first approach, the Java code obtains the sensitive data, then invokes JNI method to process the data, and finally leaks out the returns of the JNI method. In the second approach, the Java code also collects the sensitive data and invokes a JNI method to process it. After that, the Java code invokes another JNI method to obtain the results from the native contest, and leaks the results in the Java context. Note that TaintDroid cannot capture the second approach of *P2*, because it just propagates the taint tags from the parameters to the results of the JNI method whereas the JNI method for processing data and the JNI method for obtaining the results are different in the second approach. Similarly, Since Androidperf [20] is developed on top of TaintDroid, although it supports tracking information through JNI bridge, it only supports DVM. Since the *taint tracker* of Droidscope is not released, we cannot get the details of the *taint tracker*.

TaintART [9] and ARTist [16] were recently proposed to conduct taint analysis on Android running ART. However, they can only handle the information leak of *P5*, because they need to insert additional taint propagation instructions into the Java code by modifying `dex2oat`. Moreover, the new system Malton [17], which can track information flow in

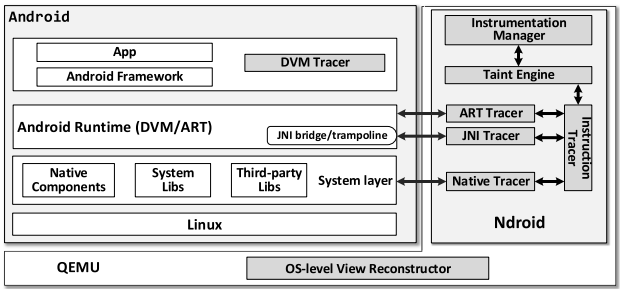


Fig. 4. NDroid architecture.

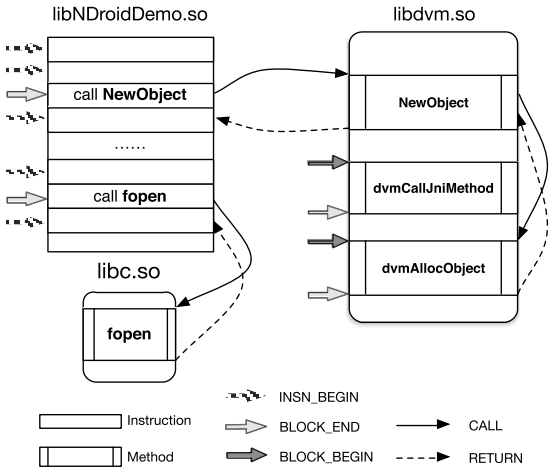


Fig. 5. Instrumentation manager.

Android apps, only supports the ART runtime and just tracks information leakage in one process. Although TaintMan [21] supports both ART and DVM, it needs to insert additional instructions into the target apps through repackaging, and thus it can neither handle packed Android apps nor track information flow in native libraries.

To address the limitations of these systems, we proposed and implemented a novel taint analysis system NDroid, which tracks information leakage in multiple contexts (i.e., Java context and native context) and supports all the aforementioned six basic scenarios of information leakages in Android system.

IV. The High-Level Design of NDroid

Each Android app runs on the top of a modified Linux kernel, with the support of Android application framework, and the Android platform contains a set of system libs. Fig. 4 illustrates the architecture of NDroid, a virtualization-based dynamic taint analysis system. QEMU is an open-source machine emulator [22], through which we can instrument instructions at translation phase and monitor each machine instruction at execution phase. To track information flows through multiple contexts, NDroid introduces six major modules into QEMU including: (1) the instrumentation manager controls when and how we do instrumentation on different modules (the JNI bridge, system libraries and apps' own native libraries) at translation phase; (2) the JNI tracker deals with the taint propagation when JNI APIs and related functions are called; (3) the instruction tracer parses each executed instruction (ARM/Thumb/Thumb2) and propagates taint information according the semantics of these instructions; (4) the native tracker conducts the taint propagation of the compiled code, and all these three handlers complete taint propagations with interfaces provided by (5) the taint engine which also maintains taint storages for both registers and memories. Moreover, (6) we employ TaintDroid as the taint tracer (i.e., DVM tracer) in DVM and implement an ART tracer module to trace the taint information in ART runtime. Note that, as NDroid currently supports both DVM and ART runtime, there are differences between the implementations on DVM and ART. We will detail the implementation of NDroid in the following subsections.

NDroid also contains a customized OS-level view reconstructor motivated by DroidScope for obtaining the information of processes and memory map in Linux. Therefore, our instrumentation manager can selectively instrument specific processes and modules. What's more, since TaintDroid carefully handles the taint propagation in the framework and DVM, we reuse the modules modified by TaintDroid and keep the taints added by NDroid consistent with TaintDroid's format so that they can work together smoothly.

V. NDroid for DVM

This section details the implementation of our system on DVM, the default Android runtime below version 5.0.

A. Instrumentation Manager

When an app sends sensitive data to its own native code by invoking native methods, the data first goes through the JNI bridge before it steps into native codes, then native codes will handle the data and possibly invoke system library calls. Therefore, the JNI bridge, apps' third party native libraries and system libraries must be instrumented in order to trace information flows through JNI.

As shown in Fig. 5, for an app's own native code (i.e., *libNDroidDemo.so*), the instrumentation manager instruments it at two different levels: (1) basic block level (i.e., indicated by BLOCK_END arrow) – if a block of code ends at invoking system library method or JNI API call, we do instrumentation at the end of it; (2) instruction level (i.e., indicated by INSN_BEGIN arrow) – each instruction is instrumented before being executed. By doing so, whenever an app's native code calls system library methods and JNI APIs we are interested in (e.g., *open(), NewObject()*, etc.), we can conduct analysis before and after they are invoked. Note that system libraries and JNI bridge are not instrumented all the time. Instead, we only instrument them when they are used by an app's own native code. However, certain methods (e.g., *dvmCallJniMethod(), dvmAllocObject()*, etc.) related to JNI bridge are instrumented at both beginnings/ends of their first/last basic blocks. Details about these methods will be discussed in Section V-B.

It is necessary to know the offsets of the methods that need instrumentation. Since it is time-consuming to calculate

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

Listing 1. 'dvmCallJNIMethod'.

```
1  typedef struct _SourcePolicy{
2      int   method_address;
3      int   tR0, tR1, tR2, tR3;
4      int   stack_args_num;
5      int*  stack_args_taints;
6      char* method_signature;
7      int   access_flag;
8  } SourcePolicy;
```

Listing 2. 'SourcePolicy'.

those offsets manually, we prepare scripts to disassemble libraries (e.g., *libc.so, libm.so, libdvm.so*, etc.), extract offsets, and generate template codes for handlers in following subsections.

B. JNI Tracer

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). It is non-trivial 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 into the native context, thus failing to track such information flows. To address this issue, the JNI bridge handler deals with instrumented JNI APIs and relevant functions, through which information flows across 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) method calling; (3) object/string/array operation; (4) field access; and (5) exception, each of which is detailed as follows.

1) JNI Entry: This category is supposed to include functions facilitating Java codes to invoke native methods. However, since JNI does not provide such interfaces, we analyzed the process of Java codes calling native methods and found that before DVM hands over control to native methods, it calls *dvmCallJNIMethod()* (as listed in 1) to do preparations. By hooking this method, we locate the parameters and their taints according to the first parameter "args", which is the frame pointer described in Fig. 1. Moreover, we identify the native method's address, access flag and signature through the third parameter "method", which points to a `Method` structure. We define a customized structure `SourcePolicy` (as shown in Listing 2) to record information of the native target: *method_address* indicates the address of the native method's first instruction; *tR0 - tR3* stores the taints of the first four parameters in registers R_0-R_3; *stack_args_num* is the number of remaining parameters on stack; *stack_args_taints* stores taints of the parameters on stack. Note that the ARM 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_signature* 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".

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

Call**Type**Method() CallNonvirtual**Type**Method() CallStatic**Type**Method()	***dvmCallMethodV()***	***dvmInterpret()***
Call**Type**MethodV() CallNonvirtual**Type**MethodV() CallStatic**Type**MethodV()	***dvmCallMethodV()***	
Call**Type**MethodA() CallNonvirtual**Type**MethodA() CallStatic**Type**MethodA()	***dvmCallMethodA()***	

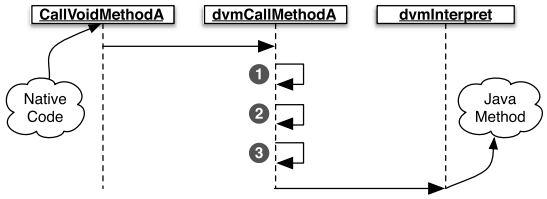


Fig. 6. Native Code Call Java Method: ① allocate a Java Method Frame; ② initialize stack with parameters and clear taint slots; ③ convert indirect inferences to real object addresses.

Therefore, by hooking *"dvmCallJNIMethod()"*, we allocate a `SourcePolicy` for each native method to be executed and we use a hash map to store the pairs of `< addr, SourcePolicy>`, where the key *"addr"* is the native method's address. Once the instruction locates at *"addr"* is being executed, `NDroid` initializes corresponding registers and memories with proper taint values according to the `SourcePolicy` paired with *"addr"*.

2) Method Calling: This category includes functions facilitating native codes to call Java methods. The first column of Table III lists these JNI APIs, and they will then call the corresponding methods in the second column. We use *dvmCallMethod∗()* to denote the methods in the second column. Finally, *dvmInterpret()* will be called before DVM interprets the target method. For instance, as shown in Fig. 6, when an app's native code calls a Java method by invoking *CallVoidMethodA()*, the following steps will be conducted: (1) allocating a Java method frame on the DVM stack (as depicted in Fig. 1); (2) putting the parameters onto the stack and clearing the taint slots; (3) scanning parameters and converting the indirect references to real object addresses. After that, *"dvmInterpret()"* is called to hands over control to the Java method.

Note that as *"dvmCallMethod*()"* methods clear the slots used to store taint tags in Java method frames, we instrument *"dvmInterpret()"* to set taint tags just before Java methods run. Moreover, since we cannot get indirect references of reference type parameters through instrumenting *"dvmInterpret()"* (③ in Fig. 6) and indirect references are used

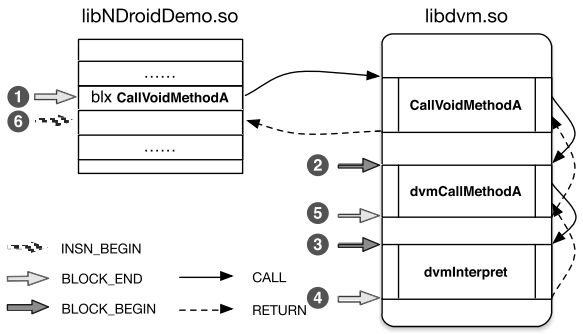


Fig. 7. Multilevel Hooking ① call *CallVoidMethodA()* ② *dvmCallMethodA()* is called ③ *dvmInterpret()* is called ④ *dvmInterpret()* return ⑤ *dvmCallMethodA()* return ⑥ return from *CallVoidMethodA()*.

in our taint storage of native context, we also instrument *"dvmCallMethod*()"*.

The overhead will be high if we instrument these functions whenever they are called, because methods "*dvmCallMethod*()*" and "*dvmInterpret()*" may also be triggered by other codes rather than the native codes under investigation. To address this issue, we propose a multilevel hooking technique to assure that the instrumentation of "*dvmCallMethod*()*" and "*dvmInterpret()*" is performed only when they are triggered by the native codes of interest. Its basic idea is to define and check a sequence of preconditions before instrumenting certain methods.

We use the method "*dvmCallVoidMethodA()*" as an example to explain the multilevel hooking technique, as shown in Fig. 7. We define six thread-specific conditions T_1, $T_2 \ldots, T_6$ to determine whether the corresponding steps of instrumentation in Fig. 7 can be performed. Let $\boldsymbol{I}_{curr}$ represents the address of the instruction being executed, $\boldsymbol{I}_{from}$ indicates the last instruction executed and $\boldsymbol{I}_{to}$ denotes the target address of branch instructions:

1) T_1 is true if $\boldsymbol{I}_{curr}$ 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, $\boldsymbol{I}_{curr}$ equals the start address of *"dvmCallMethodA()"* and $\boldsymbol{I}_{from}$ is within *"CallVoidMethodA()"*.
3) T_3 is true if T_2 is true, $\boldsymbol{I}_{curr}$ equals the start address of *"dvmInterpret()"* and $\boldsymbol{I}_{from}$ is within *"dvmCallMethodA()"*.
4) T_4 is true if T_3 is true, $\boldsymbol{I}_{curr}$ equals the end address of *"dvmInterpret()"* and $\boldsymbol{I}_{to}$ is within *"dvmCallMethodA()"*.
5) T_5 is true if T_2 is true, $\boldsymbol{I}_{curr}$ equals the end address of *"dvmCallMethodA()"* and $\boldsymbol{I}_{to}$ is within *"CallVoidMethodA()"*.
6) T_6 is true if T_1 is true, $\boldsymbol{I}_{curr}$ is within the native code and $\boldsymbol{I}_{from}$ equals the end address of *"CallVoidMethodA()"*.

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

3) Object/String/Array Operation: JNI provides interfaces for native codes to do object/string/array operations. For example, native codes can create new object/string/array through JNI functions listed in the first column of Table IV, which are denoted as **NOF**. These functions will invoke the corresponding methods in the second column of Table IV, which are denoted as **MAF**. **MAF** allocates memory for objects, strings or arrays. 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**. Moreover, other JNI APIs related to object/string/array operations are also hooked, such as functions of global and local references, functions about releasing string and getting/setting array, etc.

TABLE IV
JNI – CREATE NEW OBJECT

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

TABLE V
JNI METHODS TO GET/SET FIELD. **TYPE** ∈ {OBJECT, BYTE, SHORT, INT, LONG, FLOAT, DOUBLE, BOOLEAN, CHAR}

Get Field Functions	**Set Field Functions**
*Get**Type**Field()*	*Set**Type**Field()*
*GetStatic**Type**Field()*	*SetStatic**Type**Field()*

4) Field Access: Since native codes can access a Java object's fields through the functions listed in Table V, 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.

5) 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

For tracing information flows in apps' native codes, the native instructions are instrumented by the instrumentation manager, and the instruction tracer carries out the taint propagation of each instruction before it is executed. When an

TABLE VI

TAINT PROPAGATION LOGIC FOR ARM/THUMB/THUMB2 INSTRUCTIONS: SYMBOL "$\circ$" INDICATES BINARY OPERATORS AND SYMBOL "$\sim$" INDICATES UNARY OPERATORS; FOR $M[addr : addr + n]$ IN LDR* AND STR*, n CAN BE 1, 2 OR 4; $f(regList)$ COUNTS NUMBER OF 1 IN REGLIST

Insn Format	Taint Propagation	Description
binary-op $R_d, R_n, R_m[, \#Imm]$	$t(R_d) = t(R_n) \mid t(R_m)$	set R_d's taint with union of R_n's and R_m's
binary-op $R_d, R_n[, \#Imm]$	$t(R_d) \mid= t(R_n)$	add R_n's taint to R_d
binary-op R_d, R_n, R_m, R_a	$t(R_d) = t(R_n) \mid t(R_m) \mid t(R_a)$	set R_d's taint with union of R_n's, R_m's and R_a's
binary-op R_dHi, R_dLo, R_n, R_m	$t(R_dHi) = t(R_n) \mid t(R_m)\ t(R_dLo) = t(R_n) \mid t(R_m)$	set both R_dHi's and R_dLo's taints with union of R_n's and R_m's
unary-op R_d, R_m	$t(R_d) = t(R_m)$	set R_d's taint with R_m's
mov $R_d, \#Imm$	$t(R_d) = TAINT\_CLEAR$	clear R_d's taint
mov R_d, R_m	$t(R_d) = t(R_m)$	set R_d's taint with R_m's
LDR* $R_t, R_n, \#Imm$	$t(R_t) = t(M[addr : addr + n]) \mid t(R_n)$	set R_t's taint with union of R_n's and $M[addr : addr + n]$'s
LDR* R_t, R_n, R_m	$t(R_t) = t(M[addr : addr + n]) \mid t(R_n) \mid t(R_m)$	set R_t's taint with union of R_n's, R_m's and $M[addr : addr + n]$'s
LDM(POP) $R_n, regList$	$t(\{R_i, ..., R_{i+f(regList)-1}\})$ $= t(R_n) \mid t(M[startAddr : endAddr])$	set R_i's taint with union of R_n's and $M[startAddr : startAddr + 4]$'s, set R_{i+1}'s with union of R_n's and $M[startAddr + 4 : startAddr + 8]$'s, ..., set $R_{i+f(regList)-1}$'s with union of R_n's and $M[endAddr - 4 : endAddr]$'s
STR* $R_t, R_n, \#Imm$	$t(M[addr : addr + n]) = t(R_t)$	set $M[addr : addr + n]$'s taints with R_t's
STR* R_t, R_n, R_m	$t(M[addr : addr + n]) = t(R_t)$	set $M[addr : addr + n]$'s taints with R_t's
STM(PUSH) $regList$	$t(M[startAddr : endAddr])$ $= t(\{R_i, ..., R_{(i+f(regList)-1)}\})$	set $M[startAddr : startAddr + 4]$'s taints with R_i's, set $M[startAddr + 4 : startAddr + 8]$'s with R_{i+1}'s, ..., set $M[endAddr - 4 : endAddr]$'s taint with $R_{(i+f(regList)-1)}$'s

TABLE VII

MODELED STANDARD METHODS

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

TABLE VIII

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

instruction is fed into the instruction tracer, it first parses the semantics of the instruction, and then propagates the taint tags of the operands according to the semantics of the instruction. Note that the instruction tracer is implemented based on *darm* [23], which is a light-weight and efficient ARMv7/Thumb/Thumb2 dissembler, and we extend *darm* to enable it to propagate taint tags according to the semantics of the instruction during parsing. We list the taint propagation logic for the general types of ARM/Thumb/Thumb2 instructions that can affect taint propagation in Table VI.

D. Native Tracer

Apps' own native codes frequently call system library methods, and it will cause high performance overhead if we instrument each instruction of those methods. Therefore, we model the taint propagation of the selected system library methods, which are listed in Table VII and Table VIII. Moreover, we prepare scripts to generate template codes for all system library methods, which makes it easier for users to model taints propagation of certain methods have not been implemented yet. More precisely, they just need to enable the instrumentation of those methods and implement the taint propagation model with the help of APIs provided by the taint engine. Using the function *memcpy()* as an example, Listing 3 illustrates how to model its taint propagation operation.

```
1  //void *memcpy(void *dest,const void *src,
       size_t)
2  void memcpy_handler(TrustCallPolicy* policy,
3       CPUState* env, int isBegin){
4    if(isBegin){
5      int destAddr = env->regs[0];
6      int srcAddr = env->regs[1];
7      int nBytes = env->regs[2];
8      int i = 0;
9      for(; i < nBytes; i++){
10       //propagate the srcAddr's taint to destAddr
11       setTaint(destAddr+i, getTaint(srcAddr+i));
12     }
13   }
14 }
```

Listing 3. Handler of function '*memcpy*'.

Note that Table VIII lists the important system library methods (e.g., file read/write, network operation, etc.), including those sink methods with $*$. The system library handler records more detailed information when these methods are called, and it regards the situations as a possible information leakage when the tainted data reaches sink methods.

E. Taint Engine

The taint engine maintains shadow registers and a taint map to store registers' taints and memories' taints respectively. The taint granularity is byte-level. That is, for each byte, there is a taint flag marking it tainted or not. Moreover, if certain tainted operand is used as the memory address, we also taint the memory at the address.

The taint engine provides flexible interfaces for other modules to set/get taints easily, such as interfaces (e.g., *setMem2ToReg(), setMem4ToReg()*, etc.) for setting certain register's taint with taints got from multiple memory addresses.

1) Taint Protection: The taints for apps' native codes are stored in the QEMU process, and hence they are transparent for apps. However, malicious native codes can easily access DVM stack and heap to override taints for Java codes. One of the reasons why TaintDroid disables apps to load third party native libraries is to protect the taints on DVM stack and heap. In order to avoid introducing vulnerability by our extension

to TaintDroid, we design a practical approach to protect taints maintained by TaintDroid.

An intuitive and fine-grained method to protect those taints is to firstly label the memories allocated for taints on DVM stack and heap and then monitor native codes' accesses to those labeled memories. This method will incur unacceptable overhead because TaintDroid directly stores taints interleaved with related values instead of allocating special memory spaces for taints so that NDroid has to perform heavy-weight instrumentation for locating the memories allocated for taints. Therefore, we adopt a more practical and coarse-grained approach to protect taints for Java codes – forbidding native codes to directly access or call system library methods to indirectly access DVM stack and heap. This approach works because benign native codes will never write/read DVM stack and heap neither directly nor indirectly.

To monitor native codes' accesses to DVM stack and heap, it is necessary to get relevant address spaces. We get the stack's start address and stack size through reading the fields *"interpStackStart"* and *"interpStackSize"* stored in current thread, and hence obtain the stack's address space as $[StackStart - StackSize, StackStart]$. The DVM heap's address space can be obtained by looking for the memory module with name *"/dev/ashmem/dalvik-heap"*, because it is specifically allocated for heap using. Note that the static fields' taints are stored in the same memory space used for class definition, which is not located in DVM stack or heap. Instead, we can obtain this address space by looking for the memory module named *"/dev/ashmem/dalvik-LinearAlloc"*.

With these sensitive address spaces containing Java codes' taints, we forbid apps' native codes to access them by (1) monitoring each instruction's execution and (2) hooking system library methods related to memory reading/writing.

VI. NDroid for ART

To improve the performance of Android system, Android has replaced DVM with ART. ART appears as an alternative runtime in Android 4.4 and becomes the default runtime since Android 5.0. Instead of executing Dalvik instructions, ART compiles the Dalvik bytecode of apps into native code during the installation and then runs them directly [24]. Hence, ART may render NDroid useless because it is quite different from DVM. In the following, we will illustrate our extensions to support the ART runtime in NDroid. To make the description more clear, we denote it as NDroid-ART in this section.

A. Thread-Level View Reconstructor

NDroid-ART supports tracing taint propagations among different threads of a process simultaneously by monitoring the creation of a new thread and capturing the switch operation between different threads. More precisely, NDroid-ART instruments the kernel function *do_fork()* because it must be called when a new thread is created. As to thread switch, the kernel function *_switch_to()* realizes the task scheduler in Linux kernel. Through monitoring the invocation of this function and getting its arguments and return value, NDroid-ART can locate and fetch the *task_struct* of the thread being switched to from the kernel's memory space. Then, NDroid-ART decides whether this thread is being traced or not. If that is the case, NDroid-ART monitors this thread.

Different threads in one process share the same memory space except registers, stacks and other non-shared resources. Therefore, it is necessary to separate the taints in different threads for correct taint analysis. NDroid-ART uses thread local storage (TLS) to achieve it. A self-defined structure, named *tls_taint*, is attached to each traced thread, which records important information for taint propagation in each thread, such as ARM registers, etc. NDroid-ART also records information of the current executing method, such as whether it is a native method, its return value, invocation type and so on, because the information is bound to the method and the same method can be invoked in different threads. When a thread is traced, some global variables will point to the thread's local storage so that the private taint data cannot be interrupted by other threads' taint propagation.

It is worth noting that although DroidScope collects thread information, it does not separate the taint information in different threads, thus taints in different threads may interfere with each other [10]. Moreover, DroidScope monitors the change of the page based address register to detect the switch of processes but the switch of lightweight processes (i.e. threads) has no effect on the page based register.

B. Taint Propagation Through JNI

It is challenging to conduct dynamic taint analysis on JNI methods due to the complex procedure of invoking JNI methods in ART. Each native method declared in Dex file is compiled to a set of instructions in ART, which is called a stub. These instructions are responsible for initialing JNI environment, calling the real method in so files, and cleaning JNI environment. When the stub method calls the real implementation for the first time, it will invoke a trampoline method that first loads the real implementation and then jumps to it. When the real implementation is finished, the program returns back to the stub directly. After that, the stub calls the real implementation directly.

Native methods call Java methods through APIs like *CallTYPEMethod()*, where *TYPE* represents the type of the return value of the Java method [12], such as, "Object", "Boolean", etc. After the class and methodID are obtained through APIs *FindClass()* and *GetMethodID()*, the target Java method can be called. After the invocation of *CallTYPEMethod()* series APIs, the ArtMethod object, which represents the target Java method, can be obtained. Finally, the *ArtMethod::Invoke()* method is called to invoke the real Java method. When the target Java method is finished, the program returns back to *CallTYPEMethod()*. Finally it goes back to the so file.

To accomplish the taint propagation between native methods and Java methods, NDroid-ART instruments the key points of the JNI invocation procedure. For propagating taints from Java context to native context, NDroid-ART instruments the positions including (1) before the invocation of

the native method, (2) before the execution of the native code related to the native method, (3) after the execution of native code, and (4) after the invocation of the native method. For conveying taints from native context to Java context, NDroid-ART hooks the positions including (1) before the invocation of *CallTYPEMethod()* series APIs, (2) before the invocation of the target Java method, (3) after the invocation of the target Java method, and (4) after the invocation of *CallTYPEMethod()* series APIs.

C. ART Tracer

As TaindDroid does not support ART runtime, we implement the component ART tracer in NDroid-ART to conduct taint propagation for the ART methods. Since instruction-level taint propagation will introduce heavy overhead, we employ function-level taint propagation when analyzing Android framework APIs to improve performance. The whole procedure consists of three steps. First, for each framework API, we model the taint propagation relationship among its parameters and return value. Second, we identify and recognize each framework API to be invoked. More precisely, by performing instrumentation after each basic block is executed, NDroid-ART obtains the address of the next basic block to be executed, and then uses the address to determine the API. Third, we design a general hook function which takes in framework APIs and their taint propagation models. This function will be called before and after invoking each framework API, hence NDroid-ART can conduct the taint propagation.

Before modeling the taint propagation behavior of framework APIs, we extract their Dalvik instructions from *system@framework@boot.oat* and convert them into Java bytecode using the tool dex2jar. We built a static analysis tool to construct the control flow graph (CFG) and obtain a set of executable paths associated with parameters for each framework API. The tool then performs data flow analysis to determine whether the taint(s) will be propagated from one or more parameters to other parameters and/or the return value. If so, the taint propagation relationship among parameters and return value are recorded in a map data structure denoted as $Taint_{FA}$.

In order to improve the performance of the static taint analysis, we develop a tool to obtain the offset addresses of the functions in the app's Oat file and the Android framework (boot.oat) by parsing Oat files of both the target apps and the Android framework, and put these addresses into a configuration file. When performing instrumentation, NDroid-ART will check whether an instruction belongs to the relevant functions according to its address. If that is the case, the taint analysis will be performed.

Once a framework API is called, we obtain its taint propagation model from $Taint_{FA}$, and then propagate the taint according to the model directly, thus shortening the processing time. We acknowledge that such models are not very precise because they only consider the relationships among parameters and return values. However, creating more accurate models for framework APIs is out of the scope of this paper. Besides, NDdroid-ART also has the capacity of conducting instruction level taint propagation for the compiled ART methods. Hence, to improve the accuracy of taint propagation, users can specify NDdroid-ART to propagate the taint tags in instruction level for the ART methods.

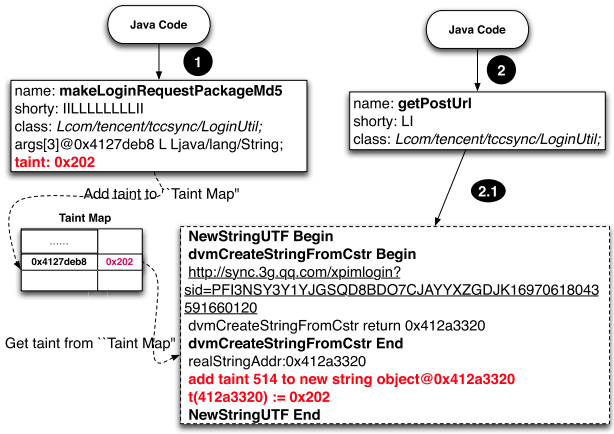


Fig. 8. Log of QQPhoneBook.

VII. Evaluation

NDroid is implemented in QEMU. NDroid for DVM (i.e., NDroid-DVM) runs Android 4.3 and NDroid for ART (i.e., NDroid-ART) runs Android 5.0. NDroid-DVM tracks information flow in the Java context through TaintDroid, and NDroid-DVM focuses on tracking the information flows through JNI. We use two real apps (i.e., QQPhoneBook 3.5 and ePhone 3.3.3) to evaluate NDroid-DVM. Since TaintDroid does not support taint propagation in ART runtime, NDroid-ART propagates taint tags in both Android framework and JNI bridge. We employ two real malware (i.e., SpyBubble and PlusLock) to evaluate NDroid-ART. Moreover, we evaluate the performance of NDroid-DVM and NDroid-ART using CaffeineMark [25]. This section answers the following questions.

- **RQ1:** Can NDroid-DVM track the information leaked through JNI bridge in DVM runtime?
- **RQ2:** Can NDroid-ART propagate taint information through multiple threads?
- **RQ3:** Can NDroid-ART track the information leakage through JNI bridge in ART runtime?
- **RQ4:** Do NDroid-DVM and NDroid-ART have reasonable performance during taint propagation?

A. Experiments on DVM

1) QQPhoneBook: NDroid-DVM discovers that QQPhoneBook 3.5 can send sensitive information related to contacts and SMS to a server named "info.3g.qq.com". Fig. 8 shows the major functions in the information flow identified by NDroid, which is an example of leaking sensitive information through pattern *P2*. In the first step, by invoking the native method "*makeLoginRequestPackageMd5()*", the Java

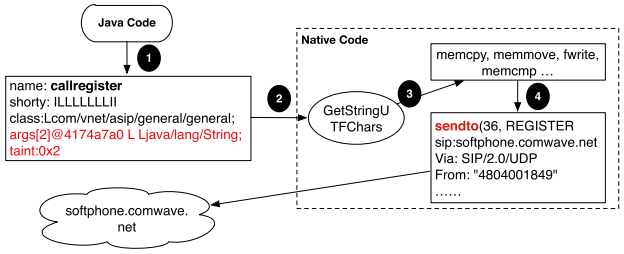


Fig. 9. Log of ePhone.

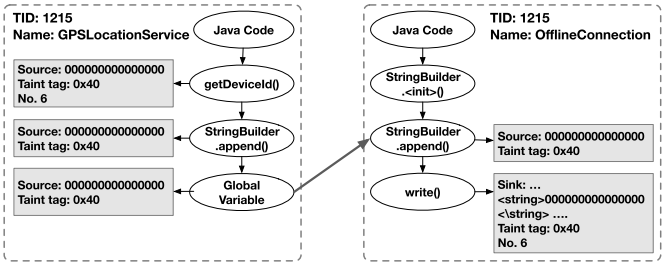


Fig. 10. Taint propagation through multiple threads.

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 "0×202". NDroid-DVM creates an entry in the taint Map to associate the memory address 0x4127deb8 with the taint "0×202".

Then, the Java code calls another native method "*getPostUrl()*" (i.e., step 2) with parameters that do not have taints. "*getPostUrl()*" invokes "*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 eventually send out the sensitive data. NDroid-DVM 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 identify such information leakage, because it does not taint the new String object and the return value of "*getPostUrl()*".

2) ePhone: NDroid-DVM finds that ePhone sends contacts related information to a name named "*softphone.comwave.net*". Fig. 9 shows the major functions in the information flow tracked by NDroid-DVM. The ePhone's Java code first calls a native method *callregister()* that passes tainted information related to contacts to its native code. Then, the native code converts the tainted Java string to C string through the method "*GetStringUTFChars()*" and further invokes many system calls (i.e., *memcpy()*, *memmove*, *fwrite*, etc.) to process the tainted information. Finally, it invokes *sendto()* to send the tainted information to the server.

Answer to **RQ1:** The experimental result shows that NDroid-DVM can successfully track the information leaked through JNI bridge in DVM runtime.

B. Experiments on ART

1) SpyBubble: Spybubble [26] is a malicious app that steals and leaks the sensitive information, such as device id, phone number, geographical location, etc. Besides, Spybubble gathers and leaks sensitive information in different threads. Fig. 10 shows the details of aforementioned malicious behaviors performed by Spybubble.

When Spybubble starts, a service named "GPSLocationService" is created to collect phone's device ID through invoking Android framework API *getDeviceId()*, which is specified as the taint source by the NDroid, When the result (i.e., device ID) of *getDeviceId()* is returned, it is attached with specific taint tag by NDroid. The tainted device ID is encoded into an XML-liked string and stored into a global variable. In addition, another service named "OfflineConnection" of

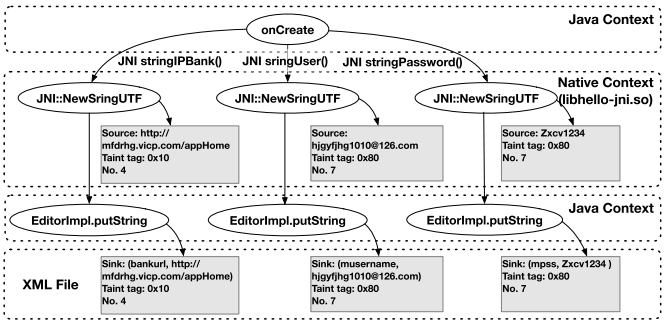


Fig. 11. Taint propagation through JNI.

Spybubble periodically reads the XML-liked string stored in the global variable, and writes the string into a hidden file named ".testFile.txt". That is, the sensitive information (i.e., device ID) is obtained and leaked by the services "GPSLocationService' and "OfflineConnection" respectively, Answer to **RQ2**: The experimental result shows that NDroid-ART can discover the information leakage flow through multiple threads.

2) PlusLock: PlusLock [27] is a malicious app that collects sensitive information, such as phone number, MAC address, device ID, etc., and then uploads the collected information to its own server or Email account. Fig. 11 depicts three JNI related malicious behaviors performed by PlusLock.

Once the main activity of PlusLock is launched, three JNI methods, i.e., *stringIPBank()*, *stringUser()* and *stringPassword()*, are called to retrieve a suspicious website address, the username of an Email account and the password of this account, respectively. In this experiment, we configure these three types of sensitive information with different taint tags (i.e., 0×10, 0×80 and 0×100). Note that, these three types of information are stored in three variables in the native code (i.e., *libhello-jni.so*) of this app. Besides, each of these three variables is passed to JNI API *NewStringUTF()* as its parameter, and then converted into a *java.lang.String* object. Thus, the return values of these three JNI methods are attached with the corresponding taint tags. All these string objects are stored in an instance of *SharedPreferences*. After that, the website address is used to establish a connection with its own server (i.e., http://mfdrhg.vicp.com) for uploading sensitive information. We also find that this malware can leak the user's phone number through Email. For example, as shown in Fig. 11, "hjgyfjhg1010@126.com" and "zxcv1234" are the username and password of the Email account, respectively.

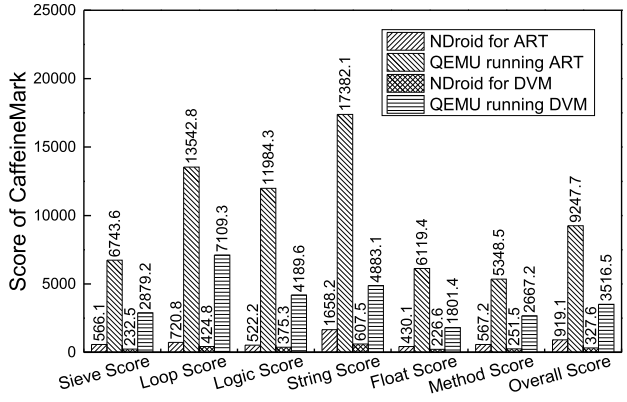


Fig. 12. Performance of NDroid.

Answer to **RQ3**: The experimental result shows that NDroid-ART can track the information leakage through multiple contexts.

C. Performance of NDroid

To evaluate the performance of NDroid, we run CaffeineMark 30 times with NDroid, and QEMU respectively. In average, compared with Qemu, the NDroid incurs 10.7 and 10.1 times overall slowdown (i.e., Fig. 12) to the DVM runtime and ART runtime respectively. Since DroidScope brings at least 11 times slowdown, NDroid introduces less slowdown than DroidScope that only supports DVM runtime.

Answer to **RQ4**: As a dynamic taint analysis tool, NDroid has reasonable performance during taint propagation on both DVM and ART runtime.

VIII. Related Work

Only a few existing systems take into account the native libraries in Android apps. Some of them dynamically collect system calls through system call hijacking [28], [29] or tools like ptrace [30], [31], strace [32], and ltrace [4]. The sequence of system calls, along with other function calls within DVM, could be used to characterize an application's behavior [33]. Based on QEMU, CopperDroid combines system calls and Android specific behaviors observed from binder to detect malware [34]. Fedler *et al.* proposed measures to control the execution of native code on the Android platform [35]. Sun *et al.* developed NativeGuard that controls the native codes of an app by forcing them to run in a non-privileged application [36]. Appcage sandboxes the native libraries to confine the app's behavior [37], but it does not track the information flow inside native libraries. 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], [6], [38]. For example, DroidNative uses specific control flow patterns to detect mobile malware using native code [6]. However, static analysis is usually hindered by various obfuscation techniques [39]–[41].

Orthogonal to monitoring functions calls, information flow tracking empowers users to understand how a program processes tainted data [33]. There are two pioneering systems for this purpose: TaintDroid [8] and DroidScope [10]. TaintDroid modified DVM to carry out dynamic taint analysis and introduces low performance overhead. However, as illustrated in Section VII, it under-taints information flows through JNI. AppFence is based on TaintDroid and does not process third-party native libraries [42]. 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]. Similarly, Androidperf [20] is implemented based on TaintDroid, although it supports tracking information through JNI bridge, it also only supports DVM platform and requires modification of the Android framework. We identify the information flows missed by these systems, and NDroid can capture them with much lower overhead than DroidScope. A recent system called Malton [17] tracks information flow in Android apps. However, it only supports ART and cannot track the information leakage crossing multiple processes.

The majority of existing systems for analyzing Android apps do not consider native libraries [43]–[47]. Instead, they usually inspect required permissions [2], [48]–[51], invoked APIs [2], [52]–[54], information flows in Java code [55], [56], and related textual information [57]–[61]. For example, WoodPecker conducts intra-procedural path-sensitive static analysis to identify capability leaks in applications of several stock Android smartphones [62], while CHEX employs inter-procedural, flow- and context-sensitive analysis to discover component hijacking vulnerabilities in applications [55].

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 [63] and designed sandbox to enable trustworthy execution of native codes [64]. Jinn defines 11 finite state machines and uses them to detect interface violations related to JNI [65]. 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 [66], malware analysis [67], understanding network protocols [68], to name a few [69], [70]. Despite many dynamic taint systems have been designed for either binary executables [69], [71], [72] or managed runtime [73], there are still many open questions in dynamic taint analysis, such as conduct control flow taint and deal with implicit information flows [69], [70]. 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.

This paper is an extension of paper [74] with many new contents, which are summarized as follows. First, we conduct a new analysis of native code on Android platform. Specifically, we investigate the framework APIs of various Android systems (i.e. from Android 2.2.3 to Android 7.0), and find that more than 71% Android framework APIs involve native code in their internal implementations. We also examine much more Android apps using native code in this paper than the previous work [74]. Based on the above analysis, we summarize six basic scenarios of information leakage in Android platform in Section III-C. Second, we re-design NDroid with the support of the new Android runtime ART in Section IV, and introduce its implementation details in Section VI, whereas the work

of [74] only supports DVM runtime. Third, to evaluate the extended version of NDroid, we conduct new experiments in Section VII with three aims: (1) can NDroid track the information leakage through multiple threads (Section VII-B.1); (2) can NDroid track the information flow involving various contexts (Section VII-B.2); and (3) how is the performance of NDroid (Section VII-C). The experimental results illustrate that NDroid can efficiently and effectively track the information flows leaked through multiple threads and involving various contexts (i.e., Java context and Native contest) with reasonable performance.

IX. 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 VII have shown that simple tools like monkeyrunner [75] 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 [76].

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 [77], [78]. 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 [79]) 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 circumvention [80]. 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.

Hybrid apps [81], [82] leverage the advantages of both traditional apps written in Java (with/without native code) and web apps using various web techniques (e.g., JavaScript, HTML 5) to speed up the development for multiple platforms. They introduce the JavaScript context in WebView [83], and rely on two mechanisms provided by Android platform to support the interaction between Java code and Javascript code: callback communication and bridge communication [81]. Using the former mechanism, developers override the callback methods in WebViewClient and WebChromeClient to react on the corresponding events [84], which can be triggered by JavaScript code with additional information as methods parameters. In the latter mechanism, developers can inject a Java object to WebView through addJavascriptInterface so that JavaScript running in WebView can call the methods of this Java object [84]. Existing dynamic taint analysis systems as well as NDroid presented in this paper do not support tracking information flows going through the JavaScript context switch. Since tracking such information flows is non-trivial and deserves another paper, we here discuss the possible approaches to achieve this purpose and will investigate them in future work.

To track the information flow through the JavaScript context, we need to track the information flow outside and inside the JavaScript runtime, separately, and carefully handle the taint propagation across the boundary between the Java context and the JavaScript context. NDroid can track the information flow outside the JavaScript runtime for both DVM and ART runtime. For tracking the information flow within JavaScript runtime, we can achieve it by modifying the JavaScript interpreter in WebView or conducting JavaScript source code instrumentation [85]. To propagation taint information across the contexts, we will first hook the important functions in the callback communications (e.g., various callback methods [84]) and the bridge communication (e.g., JavaInstanceJobject::invokeMethod for executing JavaScript methods) to capture the data to be transferred across the contexts and the corresponding taint labels. Then, we will carefully map the data in one context to the variable in another context and properly set its taint label. We will develop this functionality for NDroid in future work.

X. Conclusion

We conducted a large-scale study on the usage of native libraries of Android apps, and identified several scenarios where data leakage cannot be captured by existing systems. Based on these insights, we proposed and implemented NDroid, an efficient dynamic taint analysis system for tracking information flows across different contexts. The evaluation of real apps illustrated that NDroid can effectively identify information leaks through JNI and discover polymorphic malicious apps realized by JNI with reasonable performance overheads. We will release the new NDroid in https://github.com/rewhy/NDroid.

Acknowledgements

The authors would like to thank the anonymous reviewers for their quality reviews and suggestions.

References

[1] (2017). *Mobile Operating System Market Share Worldwide*. [Online]. Available: http://gs.statcounter.com/os-market-share/mobile/worldwide

[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, pp. 50–52.

[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, pp. 281–294.

[4] M. Spreitzenbarth, F. Echtler, F. Freiling, T. Schreck, and J. Hoffmann, "Mobile-sandbox: Having a deeper look into Android applications," in *Proc. SAC*, 2013, pp. 1808–1815.

[5] X. Wei, L. Gomez, I. Neamtiu, and M. Faloutsos, "ProfileDroid: Multi-layer profiling of Android applications," in *Proc. MobiCom*, 2012, pp. 137–148.

[6] S. Alam, Z. Qu, R. Riley, Y. Chen, and V. Rastogi, "DroidNative: Automating and optimizing detection of Android native code malware variants," *Comput. Secur.*, vol. 65, pp. 230–246, Mar. 2017.

[7] V. Afonso *et al.*, "Going native: Using a large-scale analysis of Android apps to create a practical native-code sandboxing policy," in *Proc. NDSS*, 2016, pp. 1–15.

[8] W. Enck *et al.*, "TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones," in *Proc. USENIX OSDI*, 2010, pp. 1–15.

[9] M. Sun, T. Wei, and J. Lui, "Taintart: A practical multi-level information-flow tracking system for Android runtime," in *Proc. ACM CCS*, 2016, pp. 331–342.

[10] L. K. Yan and H. Yin, "DroidScope: Seamlessly reconstructing the OS and Dalvik semantic views for dynamic Android malware analysis," in *Proc. USENIX Secur.*, 2012, pp. 569–584.

[11] (2013). *Android Security Analysis Challenge: Tampering Dalvik Bytecode During Runtime*. [Online]. Available: http://bluebox.com/labs/android-security-challenge/

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

[13] (2013). *Android NDK*. [Online]. Available: https://developer.android.com/ndk

[14] E. Hughes. (2011). *JNI Local Reference Changes in ICS*. [Online]. Available: https://goo.gl/5PkWUX

[15] F. Bellard, "QEMU, a fast and portable dynamic translator," in *Proc. USENIX ATC, FREENIX Track*, 2005, pp. 1–6.

[16] M. Backes, S. Bugiel, O. Schranz, P. von Styp-Rekowsky, and S. Weisgerber. (Jul. 2016). "ARTist: The Android runtime instrumentation and security toolkit." [Online]. Available: https://arxiv.org/abs/1607.06619

[17] L. Xue, Y. Zhou, T. Chen, X. Luo, and G. Gu, "Malton: Towards on-device non-invasive mobile malware analysis for ART," in *Proc. USENIX Secur.*, 2017, pp. 289–306.

[18] N. Nethercote and J. Seward, "Valgrind: A framework for heavy-weight dynamic binary instrumentation," in *Proc. ACM PLDI*, 2007, pp. 89–100.

[19] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "PScout: Analyzing the Android permission specification," in *Proc. CCS*, 2012, pp. 217–228.

[20] L. Xue, C. Qian, and X. Luo, "AndroidPerf: A cross-layer profiling system for Android applications," in *Proc. IWQoS*, 2015, pp. 115–124.

[21] W. You, B. Liang, W. Shi, P. Wang, and X. Zhang, "TaintMan: An ART-compatible dynamic taint analysis framework on unmodified and non-rooted Android devices," *IEEE Trans. Dependable Secure Comput.*, to be published.

[22] (2013). *Qemu*. [Online]. Available: http://wiki.qemu.org/Main_Page

[23] (2017). *A Light-Weight and Efficient Disassembler Written in C for the ARMv7 Instruction Set*. [Online]. Available: https://github.com/jbremer/darm

[24] (2017). *Android Runtime*. [Online]. Available: https://goo.gl/peCC18

[25] (2017). *CaffeineMark*. [Online]. Available: http://www.benchmarkhq.ru/cm30/

[26] (2017). *SpyBubble*. [Online]. Available: http://www.prospybubble.com

[27] (2017). *Pluslock*. [Online]. Available: https://goo.gl/wdqFGd

[28] T. Bläsing, L. Batyuk, A.-D. Schmidt, S. A. Camtepe, and S. Albayrak, "An Android application sandbox system for suspicious software detection," in *Proc. MALWARE*, 2010, pp. 55–62.

[29] Y. Shao, X. Luo, and C. Qian, "RootGuard: Protecting rooted Android phones," *Computer*, vol. 47, no. 6, pp. 32–40, Jun. 2014.

[30] G. Portokalidis, P. Homburg, K. Anagnostakis, and H. Bos, "Paranoid Android: Versatile protection for smartphones," in *Proc. ACSAC*, 2010, pp. 347–356.

[31] M. Zheng, M. Sun, and J. C. S. Lui, "DroidTrace: A ptrace based Android dynamic analysis system with forward execution capability," in *Proc. IWCMC*, 2014, pp. 128–133.

[32] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani, "Crowdroid: Behavior-based malware detection system for Android," in *Proc. SPSM*, 2011, pp. 15–26.

[33] M. Egele, T. Scholte, E. Kirda, and C. Kruegel, "A survey on automated dynamic malware analysis techniques and tools," *ACM Comput. Surv.*, vol. 44, no. 2, pp. 1–49, 2012.

[34] 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, pp. 1–6.

[35] R. Fedler, M. Kulicke, and J. Schütte, "Native code execution control for attack mitigation on Android," in *Proc. SPSM*, 2013, pp. 15–20.

[36] M. Sun and G. Tan, "NativeGuard: Protecting Android applications from third-party native libraries," in *Proc. WiSec*, 2014, pp. 165–176.

[37] Y. Zhou, K. Patel, L. Wu, Z. Wang, and X. Jiang, "Hybrid user-level sandboxing of third-party Android apps," in *Proc. ACM ASIACCS*, 2015, pp. 19–30.

[38] A.-D. Schmidt *et al.*, "Static analysis of executables for collaborative malware detection on Android," in *Proc. ICC*, 2009, pp. 1–5.

[39] A. Moser, C. Kruegel, and E. Kirda, "Limits of static analysis for malware detection," in *Proc. ACSAC*, 2007, pp. 421–430.

[40] Y. Zhang, X. Luo, and H. Yin, "DexHunter: Toward extracting hidden code from packed Android applications," in *Proc. ESORICS*, 2015, pp. 293–311.

[41] L. Xue, X. Luo, L. Yu, S. Wang, and D. Wu, "Adaptive unpacking of Android apps," in *Proc. ICSE*, 2017, pp. 358–369.

[42] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall, "These aren't the droids you're looking for: Retrofitting Android to protect data from imperious applications," in *Proc. CCS*, 2011, pp. 639–652.

[43] P. Faruki *et al.*, "Android security: A survey of issues, malware penetration, and defenses," *IEEE Commun. Surveys Tuts.*, vol. 17, no. 2, pp. 998–1022, 2nd Quart., 2015.

[44] Sufatrio, D. J. J. Tan, T.-W. Chua, and V. L. L. Thing, "Securing Android: A survey, taxonomy, and challenges," *ACM Comput. Surv.*, vol. 47, no. 4, 2015, Art. no. 58. [Online]. Available: https://dl.acm.org/citation.cfm?id=2733306

[45] Y. Acar, M. Backes, S. Bugiel, S. Fahl, P. McDaniel, and M. Smith, "SoK: Lessons learned from Android security research for appified software platforms," in *Proc. IEEE Symp. Secur. Privacy*, May 2016, pp. 433–451.

[46] M. Xu *et al.*, "Toward engineering a secure Android ecosystem: A survey of existing techniques," *ACM Comput. Surv.*, vol. 49, no. 2, 2016, Art. no. 30.

[47] K. Tam, A. Feizollah, N. B. Anuar, R. Salleh, and L. Cavallaro, "The evolution of Android malware and Android analysis techniques," *ACM Comput. Surv.*, vol. 49, no. 4, 2017, Art. no. 76.

[48] H. Peng *et al.*, "Using probabilistic generative models for ranking risks of Android apps," in *Proc. CCS*, 2012, pp. 241–252.

[49] W. Enck, M. Ongtang, and P. McDaniel, "On lightweight mobile phone application certification," in *Proc. CCS*, 2009, pp. 235–245.

[50] K. Chen *et al.*, "Contextual policy enforcement in Android applications with permission event graphs," in *Proc. NDSS*, 2013.

[51] B. P. Sarma, N. Li, C. Gates, R. Potharaju, C. Nita-Rotaru, and I. Molloy, "Android permissions: A perspective combining risks and benefits," in *Proc. SACMAT*, 2012, pp. 13–22.

[52] M. Fan *et al.*, "Android malware familial classification and representative sample selection via frequent subgraph analysis," *IEEE Trans. Inf. Forensics Security*, vol. 13, no. 8, pp. 1890–1905, Aug. 2018.

[53] L. Yu, T. Zhang, X. Luo, L. Xue, and H. Chang, "Toward automatically generating privacy policy for Android apps," *IEEE Trans. Inf. Forensics Security*, vol. 12, no. 4, pp. 865–880, Apr. 2017.

[54] L. Yu, T. Zhang, X. Luo, and L. Xue, "AutoPPG: Towards automatic generation of privacy policy for Android applications," in *Proc. SPSM*, 2015, pp. 39–50.

[55] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, "CHEX: Statically vetting Android apps for component hijacking vulnerabilities," in *Proc. CCS*, 2012, pp. 229–240.

[56] C. Qian, X. Luo, Y. Le, and G. Gu, "VulHunter: Toward discovering vulnerabilities in Android applications," *IEEE Micro*, vol. 35, no. 1, pp. 44–53, Jan./Feb. 2015.

[57] R. Pandita, X. Xiao, W. Yang, W. Enck, and T. Xie, "WHYPER: Towards automating risk assessment of mobile applications," in *Proc. USENIX Secur.*, 2013, pp. 527–542.

[58] Z. Qu, V. Rastogi, X. Zhang, Y. Chen, T. Zhu, and Z. Chen, "AutoCog: Measuring the description-to-permission fidelity in Android applications," in *Proc. CCS*, 2014, pp. 1354–1365.

[59] L. Yu, X. Luo, C. Qian, and S. Wang, "Revisiting the description-to-behavior fidelity in Android applications," in *Proc. SANER*, 2016, pp. 415–426.

[60] L. Yu, X. Luo, C. Qian, S. Wang, and H. K. N. Leung, "Enhancing the description-to-behavior fidelity in Android apps with privacy policy," *IEEE Trans. Softw. Eng.*, to be published.

[61] L. Yu, X. Luo, X. Liu, and T. Zhang, "Can we trust the privacy policies of Android apps?" in *Proc. DSN*, 2016, pp. 538–549.

[62] M. C. Grace, Y. Zhou, Z. Wang, and X. Jiang, "Systematic detection of capability leaks in stock Android smartphones," in *Proc. NDSS*, 2012, p. 19.

[63] G. Tan and J. Croft, "An empirical security study of the native code in the JDK," in *Proc. USENIX Secur.*, 2008, pp. 365–378.

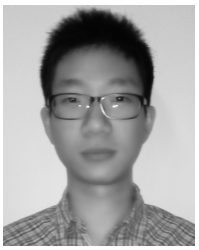
[64] M. Sun and G. Tan, "JVM-portable sandboxing of Java's native libraries," in *Proc. ESORICS*, 2012, pp. 842–858.

[65] 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, pp. 36–49.

[66] J. Newsome and D. Song, "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software," in *Proc. NDSS*, 2005, pp. 1–43.
[67] 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, pp. 116–127.
[68] G. Wondracek, P. Comparetti, C. Kruegel, and E. Kirda, "Automatic network protocol analysis," in *Proc. NDSS*, 2008, pp. 1–18.
[69] E. J. 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 Symp. Secur. Privacy*, May 2010, pp. 317–331.
[70] B. Livshits, "Dynamic taint tracking in managed runtimes," Microsoft Res., Tech. Rep. MSR-TR-2012-114, 2012. [Online]. Available: https://www.microsoft.com/en-us/research/publication/dynamic-taint-tracking-in-managed-runtimes/
[71] 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, pp. 142–154, 2011.
[72] V. P. Kemerlis, G. Portokalidis, K. Jee, and A. D. Keromytis, "Libdft: Practical dynamic data flow tracking for commodity systems," in *Proc. VEE*, 2012, pp. 1–12.
[73] V. Haldar, D. Chandra, and M. Franz, "Dynamic taint propagation for Java," in *Proc. ACSAC*, 2005, p. 311.
[74] C. Qian, X. Luo, Y. Shao, and A. T. S. Chan, "On tracking information flows through JNI in Android applications," in *Proc. IEEE DSN*, Jun. 2014, pp. 180–191.
[75] (2017). *Monkeyrunner*. [Online]. Available: https://goo.gl/x2JhA1
[76] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: An input generation system for Android apps," in *Proc. FSE*, 2013, pp. 224–234.
[77] (2013). *AppUse—Android Pentest Platform Unified Standalone Environment*. [Online]. Available: https://appsec-labs.com/AppUse
[78] L. Bordoni, M. Conti, and R. Spolaor, "Mirage: Toward a stealthier and modular malware analysis sandbox for Android," in *Proc. ESORICS*, 2017, pp. 278–296.
[79] ARM Ltd. (2017). *Trustzone*. [Online]. Available: https://goo.gl/mvH17K
[80] G. S. Babil, O. Mehani, R. Boreli, and M.-A. Kaafar, "On the effectiveness of dynamic taint analysis for protecting against private information leaks on Android-based devices," in *Proc. SECRYPT*, 2013, pp. 1–8.
[81] S. Lee, J. Dolby, and S. Ryu, "HybriDroid: Static analysis framework for Android hybrid applications," in *Proc. ASE*, 2016, pp. 250–261.
[82] X. Jin, X. Hu, K. Ying, W. Du, H. Yin, and G. N. Peri, "Code injection attacks on HTML5-based mobile apps: Characterization, detection and mitigation," in *Proc. CCS*, 2014, pp. 66–77.
[83] *Webview*. Accessed: 2017. [Online]. Available: https://developer.android.com/reference/android/webkit/WebView
[84] N. Gok and N. Khanna, *Building Hybrid Android Apps With Java and JavaScript: Applying Native Device APIs* (Japplying Native Device Apis). Newton, MA, USA: O'Reilly Media, 2013.
[85] K. Sen, S. Kalasapur, T. Brutch, and S. Gibbs, "Jalangi: A selective record-replay and dynamic analysis framework for JavaScript," in *Proc. ESEC/FSE*, 2013, pp. 488–498.

Lei Xue received the Ph.D. degree in computer science from The Hong Kong Polytechnic University. He is currently a Post-Doctoral Research Fellow with the Department of Computing, The Hong Kong Polytechnic University. His current research focuses on network security, mobile security, and network measurement.

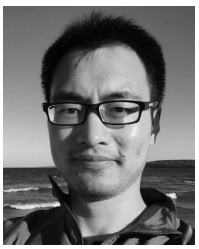
Chenxiong Qian is currently pursuing the Ph.D. degree with the School of Computer Science, Georgia Tech. He was with The Hong Kong Polytechnic University as a Research Assistant from 2013 to 2014. His research focuses on system security and privacy, and using program analysis to solve system security and privacy problems.



Hao Zhou received the B.S. and M.S. degrees from the Nanjing University of Posts and Communications. He is currently pursuing the Ph.D. degree with the Department of Computing, The Hong Kong Polytechnic University. He was with PolyU as a Research Assistant from 2016 to 2018. His current research focuses on system security, mobile security, IoT security and software testing.

Xiapu Luo received the Ph.D. degree in computer science from The Hong Kong Polytechnic University. He was a Post-Doctoral Research Fellow with the Georgia Institute of Technology. He is currently an Assistant Professor with the Department of Computing and an Associate Researcher with the Shenzhen Research Institute, The Hong Kong Polytechnic University. His current research focuses on smartphone security and privacy, network security and privacy, and Internet measurement.

Yajin Zhou received the Ph.D. degree in computer science from North Carolina State University, Raleigh, NC, USA. He is currently a ZJU 100 Young Professor with the Institute of Cyber Security Research and the College of Computer Science and Technology, Zhejiang University, China. His research mainly focuses on smartphone and system security, such as identifying real-world threats and building practical solutions, mainly in the context of embedded systems (or IoT devices).



Yuru Shao is currently pursuing the Ph.D. degree with the University of Michigan at Ann Arbor, Ann Arbor. He was a Research Assistant with The Hong Kong Polytechnic University from 2013 to 2014. His research generally focuses on network and system security, mobile security, industrial network security, cyber-physical systems security, and vulnerability discovery and analysis.

Alvin T. S. Chan received the B.Eng. degree (Hons.) from Leeds University, U.K., and the Ph.D. degree from the University of New South Wales, Australia. He had extensive industrial experience as a Research Scientist with CSIRO, Australia, and as a Program Manager with NUS. He joined The Hong Kong Polytechnic University as an Academic. He is currently with the Singapore Institute of Technology and is appointed as the Deputy Cluster Director for ICT and Program Director for the Digipen Programs. His research has produced over 120 publications in international journals and conferences. His research interests primarily focus on Internet of Things for industry, cyber security for industrial automation systems, Industry 4.0, machine-to-machine communications, middleware technologies and software engineering.