AndroidPerf: A Cross-layer Profiling System for Android Applications

Lei Xue, Chenxiong Qian, and Xiapu Luo[§]
Department of Computing, The Hong Kong Polytechnic University
The Hong Kong Polytechnic University Shenzhen Research Institute
{cslxue,cscqian,csxluo}@comp.polyu.edu.hk

***Abstract*—Profiling Android applications (or simply apps) is an important way to discover and locate various problems in apps, such as performance bottleneck, security loopholes, etc. Although many dynamic profiling systems for apps have been proposed, they are limited in dealing with the multiple-layer nature of Android and thus cannot reveal issues due to the underlying platform or poor interactions between different layers. Note that since apps usually run in Dalvik virtual machine (DVM) and each DVM is a process in Android's customized Linux kernel, a simple operation in DVM will lead to many function calls in different layers. In this paper, we propose AndroidPerf, a cross-layer profiling system, including the DVM layer, the system layer, and the kernel layer, for Android apps. It consists of one sub-system that performs cross-layer dynamic taint analysis to collect control flow and data flow information, and another sub-system that conducts instrumentation on all layers for collecting performance information. We have implemented AndroidPerf in 9,125 lines of C/C++ and 1,016 lines of Python scripts along with some modifications to Android's framework. Besides evaluating its functionality and overhead, we have applied AndroidPerf to reveal real performance issues through case studies.**

I. Introduction

Being one of the most popular mobile operating systems, Android has occupied 81.5% market share [1] and owned more than 1.5 million applications (or simply apps) in Google Play market [2]. It is not easy to develop popular apps because besides functionality and usability users and developers also care about many other features, including security and privacy [3], [4], performance [5], [6], and energy consumption [7], [8], etc. To profile apps and locate potential issues, various dynamic approaches have been proposed [5]–[18].

However, these approaches are limited in their ability to deal with the multiple-layer nature of Android, and thus cannot uncover issues due to the underlying platform or poor interactions between different layers. It is worth noting that each app usually runs in a Dalvik virtual machine (DVM) and DVM is a process in Android's customized Linux kernel. Moreover, app can invoke native methods through Java native interface (JNI) or be written in native codes [19]. Operations in DVM will lead to function calls in different layers, such as functions in Android's framework, functions in native libraries, exported system calls and internal functions in kernel. It is worth noting that hidden performance or security issues in apps often result from the subtle interactions among components on different layers [7], [14]. Unfortunately, existing dynamic profiling systems usually focus on the DVM layer with additional information collected from a few sources [9], [12], [13], [15]. A few systems consider the native libraries and system calls [7], [8], [11], and very few system takes into account the functions in Android's Linux kernel.

To fill in this gap, we propose AndroidPerf, a cross-layer profiling system for Android apps. AndroidPerf covers three layers, including the DVM layer consisting of functions in Android framework, the system layer containing system or third-party native libraries, and the kernel layer comprising exported system calls and internal functions. AndroidPerf consists of two sub-systems. One performs cross-layer *dynamic taint analysis* [20] to collect control flow and data flow information. The other conducts instrumentation on three layers to collect performance information, such as timestamp.

By labeling selected data sources, which are usually named as taint sources, a dynamic taint analysis system [15], [16], [18] can track the propagation of labelled data by enumerating functions invoked, determining whether a function or an instruction handles the labelled data or not, and tracing how the label (it is usually called as a taint) is propagated from one variable or memory space to another through different instructions until the label reaches selected destinations, which are usually named as taint sinks. It is worth noting that the fine-grained information obtained through the cross-layer dynamic taint analysis empowers users to quickly locate issues.

It is challenging to design and realize AndroidPerf because of two reasons. First, even a simple operation in the DVM layer can result in many function invocations in different layers. For example, we observed that establishing a TCP connection to a remote server in DVM will invoke 70 functions in the DVM layer, 43 functions in the system layer, and 388 functions and 9 exceptions in the kernel layer. AndroidPerf needs to track these functions and their relationships.

Second, it is not-trivial to trace the taint propagation across different layers because the function invocation styles, memory space, types, and taint propagation rules in different layers are not the same. Moreover, cross-layer operations involve special mechanisms. For example, function calls from an app running in DVM to native codes in the system layer needs JNI bridge while function calls from the system layer to the kernel layer

[§] The corresponding author.

978-1-4673-7113-1/15/$31.00

are realized by SWI (soft interrupt) [21]. Note that none of existing system supports cross-layer dynamic taint analysis.

In summary, we make the following major contributions:

1) To our best knowledge, AndroidPerf is the first cross-layer profiling system for Android applications, covering the DVM layer, the system layer and the kernel layer.
2) We have tackled several challenging issues to design and realize a cross-layer dynamic taint analysis sub-system for AndroidPerf to collect control flow and data flow information.
3) We have implemented AndroidPerf in 9,125 lines of C/C++ and 1,016 lines of Python scripts along with some modifications to Android's framework.
4) We have carefully evaluated AndroidPerf's functionality and overhead, and applied AndroidPerf to reveal real performance issues through case studies.

The rest of this paper is organized as follows. Section II and Section III describe the design and the implementation of AndroidPerf, respectively. Section IV presents the evaluation results and the applications of AndroidPerf. After describing AndroidPerf's limitations and the future work in Section V, we introduce the related work in Section VI and conclude the paper in Section VII.

II. Design of AndroidPerf

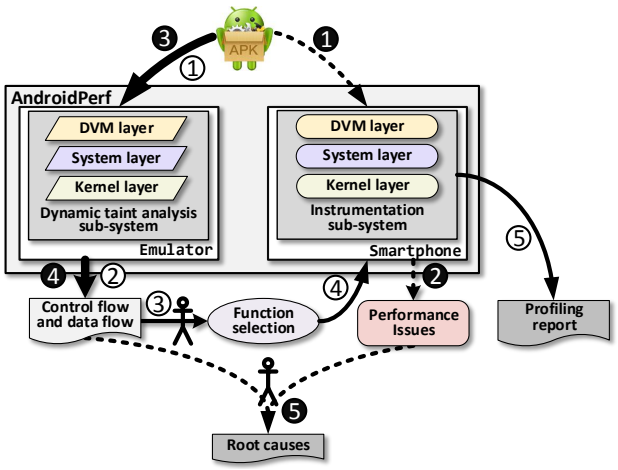


Fig. 1. Overview of AndroidPerf.

A. Overview

As shown in Fig. 1, AndroidPerf consists of two sub-systems. One performs dynamic taint analysis at three layers to generate control flow and data flow information. Control flow refers to the sequence of functions invoked by an operation, and data flow contains the functions involved in the taint propagation. Note that the functions in control flow is a superset of that in data flow. This sub-system is built on top of the open source emulator QEMU [22]. The other sub-system conducts instrumentation on three layers to collect performance information, such as timestamp, and this sub-system runs in a real smartphone.

Fig.1 also illustrates two use scenarios of AndroidPerf. Given an app, AndroidPerf can first collect all functions called by an operation and label those involved in taint propagation (i.e.,① and ②). Based on this information, a user can decide critical functions (i.e., ③) and then ask AndroidPerf to instrument them (i.e., ④) and collect run-time information (i.e., ⑤). For example, Section IV-C demonstrates how to quantify the delay introduced by each layer to packet transmission. Note that how to select critical functions is out of the scope of this paper because different functions may be selected for different purposes and we will examine it in future work.

Alternatively, a user may have observed performance issues in an app by using the instrumentation sub-system and wants to determine the root cause (i.e.,❶ and ❷). She can use the dynamic taint analysis sub-system to output the control flow and data flow information (i.e., ❸ and ❹). Such information along with the performance measurement result will help she find the root causes (i.e., ❺). For example, Section IV-B demonstrates how AndroidPerf facilitates in explaining the performance differences among three file writing approaches.

B. Dynamic Taint Analysis Sub-system

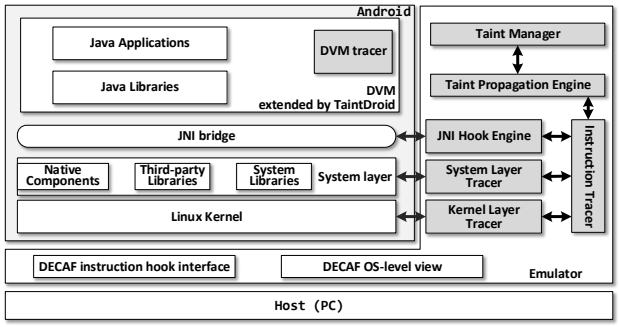


Fig. 2. Architecture of the Dynamic Taint Analysis Sub-system.

This sub-system takes in an app, runs it, and then outputs the app's control flow and data flow information. To our best knowledge, it is the first system that can perform dynamic taint analysis for all three layers, including DVM layer, system layer, and kernel layer.

We adopt the virtual machine introspection (VMI) technique [23] to develop the dynamic taint analysis sub-system. As shown in Fig.2, the Android system runs in an emulator and AndroidPerf has major components within the emulator to monitor the execution of apps and record necessary information. Since the components in the emulator collect information in the system layer and the kernel layer, AndroidPerf uses TaintDroid [15] to trace information flows within DVM. We describe the functionality of major components in the following paragraphs and detail the implementation in Section III.

DVM Tracer. It extends Android's profiling framework to automatically obtain the sequence of invoked functions. The DVM tracer overcomes the limitations in the original profiling framework. First, the DVM tracer can trace the whole life cycle of an app *without* modifying the app. Second, the information related to function entering and existing can be dumped

in real time while the original framework can only output it after the profiling process stops. Consequently, the DVM tracer can also circumvent the buffer size limitation (8MB by default) in Android's profiling framework. Moreover, DVM tracer modifies Taintdroid to dump the functions involved in taint propagation. Section III-A details its implementation.

System Layer Tracer. It traces native functions in both system libraries and third-party libraries. More concretely, it records each function's entry address and return address, and decides whether a function is called by comparing the current address with record entry addresses and return addresses. We elaborate on its realization in Section III-C.

Kernel Layer Tracer. It traces functions in kernel and deals with other issues including process switch and exception handling. The kernel functions are traced in a similar way as native functions. To monitor process switch, we hook the function *cpu_v7_switch_mm()* relevant to process switch. The kernel layer tracer also traces ARM interrupts and exceptions to identify the corresponding handling routines. Note that all interrupts (including hardware reset) are also called exceptions in ARM [21]. For example, all system calls on the system layer invoke their corresponding kernel implementations through SWI (software interrupt). Section III-D details the implementation of kernel layer tracer.

Taint Manager. Due to diverse requirements, AndroidPerf supports several types of taint sources and taint sinks through the taint manager. In the DVM layer, AndroidPerf re-uses the taint sources of TaintDroid [15]. In the system layer and the kernel layer, AndroidPerf supports four types of taint sources:

1) Taints propagated from DVM to the system layer;
2) The parameters and/or return values of native functions or kernel functions;
3) Memory. For example, to trace the taint propagation process of the *sk_buff* struct, the user can attach taint labels to its memory. Moreover, different parameters of *sk_buff* can also be attached with different taint labels.
4) I/O interfaces. Such taint sources are designed to track data (e.g., a packet) from outside. AndroidPerf regards the network interface (NIC) as a taint source. Therefore, when a packet is received by the NIC, it will be attached with taint labels automatically.

Currently, AndroidPerf supports two types of taint sinks, including NIC's sending functions and memory deallocation functions. For example, when a packet is sent out through an NIC, AndroidPerf will terminate the taint propagation of this packet. More taint sources and sinks, such as those in [24], will be added in future work.

JNI Hook Engine. It tracks the data flow through JNI bridge for propagating the taints between the DVM layer and the system layer. More precisely, the JNI Hook Engine instruments selected JNI-related functions, which are responsible for delivering data across the DVM layer and the system layer, for taint propagation. We elaborate on the implementation of JNI hook engine in Section III-B.

Taint Propagation Engine. It directs the taint propagation. In the DVM layer, AndroidPerf relies on TaintDroid [15] to trace taints. Since TaintDroid does not support taint analysis in the system layer and the kernel layer, AndroidPerf realizes an ARM ISA-level taint engine with byte granularity. Hence, *char* type occupies one taint label, *short* type occupies two taint labels and *int* type occupies four taint labels.

AndroidPerf supports three classes of taint propagation policies:

1) set operation ($t(D) = t(S)$);
2) add (or) operation ($t(D) = t(D)|t(S)$);
3) clear operation $t(D) = 0$;

where $t(S)$ represents the taint label of the source address or register and $t(D)$ represents the taint label of the destination address or register. If AndroidPerf propagates S's taint to D with set operation, D has the same taint label with S. Note that AndroidPerf supports *multiple* taints.

If the add operation is used, D has both its own taint label and S's taint label. Since each taint type occupies one bit, AndroidPerf can maintain 8 types of taints simultaneously when *byte* is used to store taint labels. It is easy to use larger data type to store more taint types.

Moreover, we add the taint clear operation in AndroidPerf to reduce over-taints. When the clear operation is applied to D, its taint label will be removed. How to realize the taint propagation engine is explained in Section III-E.

Instruction Tracer It hooks each ARM/Thumb instruction and disassembles the instruction so that the taint propagation engine can propagate the taint according the instruction logic. The instruction tracer depends on the instruction hook interface of DECAF [25] and can obtain each instruction before it is executed. Moreover, the system layer tracer and the kernel layer tracer trace function entering and exiting operations according to the destination address of the block jump instruction.

C. Constructing Control Flow and Data Flow Information.

AndroidPerf parses the tracing logs generated by the dynamic taint analysis subsystem to construct control flow and data flow information. Since the entering and the exiting operations of each function are recorded, we extract function sequences according to the execution order of functions. If one taint propagation operation occurs between the entering operation and the exiting operation of one function, it means that the taint is propagated in the function.

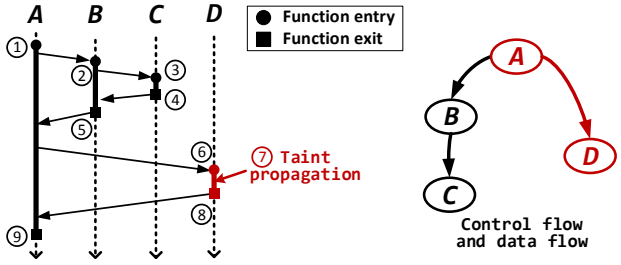


Fig. 3. Constructing control flow and data flow from tracing logs.

Fig. 3 demonstrates how to construct the control flow and data flow information for functions A, B, C and D. The

right subfigure in Fig. 3 shows the records of function calls. Function A calls function B, function B invokes function C, and function A calls function D after function B exits. We can construct the control flow as shown in the left subfigure in Fig. 3. Moreover, since the taint propagation information between function D's entry point and exit point can be loated from tracing logs, we know that function A and function D are involved in taint propagation and therefore we can obtain the data flow information marked by red nodes and red line.

D. Instrumentation Sub-system

After the critical functions are selected from the output of the dynamic taint analysis sub-system, AndroidPerf will perform instrumentation to collect required information.

As an example of demonstrating the usage of AndroidPerf, we instrument functions at different layers to collect timestamps of invoking certain functions. At the DVM layer, we extend Android's profiling framework to collect information about specified apps according to their Linux user IDs (UID). Then we parse the trace files and get the time spent on each D-VM function with microsecond resolution. In the system layer, AndroidPerf uses strace [26] to trace system calls and print the time spent on each syscall with microsecond resolution. In the kernel layer, AndroidPerf employs kprobe [27] to hook kernel functions and uses function *ktime_get()* to acquire timestamp with nanosecond resolution. The time spent on one kernel function can be calculated by using the timestamps obtained at its entry point and exit point.

III. Implementation of AndroidPerf

We have implemented AndroidPerf in 9,125 lines of C/C++ with 1,016 lines of Python scripts. Moreover, we modify Android's framework to collect control flow and data flow information and conduct instrumentation at the DVM layer. In this section, we detail the implementation of some major components in AndroidPerf.

A. DVM Tracer

We extend Android's profiling framework to collect control flow information. Note that the function *dvmMethodTraceAdd()* implemented in *Profile.c* is called each time when a method is entered or exits. This function provides class name, method name, method shorty (descriptions of the types of parameters), thread ID and process ID. Hence, we modify this function to dump control flow information.

However, Android's profiling functions are not enabled by default. They can be started or stopped by calling *Debug.startMethodTrace()* or *Debug.stopMethodTrace()* in apps. To profile apps *without* source codes, we modify *ActivityManagerProxy.attachApplication()* in *ActivityManagerNative.java* to start app profiling automatically. Note that when an app is launched, it calls *ActivityManagerProxy.attachApplication()* to pass the new *IApplicationThread* instance of the new process to *ActivictyManagerService*.

To specify the app under examination, we put its UID, which can be found from */data/system/packages.list*, in the file */sdcard/uid*. At the end of function *ActivityManagerProxy.attachApplication()*, the DVM tracer reads the UID from that file and compares it with the current app's UID. If they are the same, the app will be traced. To analyze multiple apps, users can put their UIDs in that file.

Taint propagation at the DVM layer is achieved by TaintDroid. Since TaintDroid doesn't output taint information during taint propagation, we modify TaintDroid to output the taint propagation information in real time.

B. JNI Hook Engine

JNI hook engine is implemented by hooking function *dvmCallJNIMethod()* (JNI Call Bridge) in *libdvm.so*, which is used to transfer execution from DVM to native code. More precisely, the JNI hook engine first locates the parameters passed to the native codes and their taint labels according to the first parameter of *dvmCallJNIMethod()* and then points to the memory storing the method's parameters and their taint labels. After that, the JNI hook engine extracts the method address, access flags and method shorty by parsing the third parameter of *dvmCallJNIMethod()*, which points to the memory of structure *Method*. Finally, the JNI hook engine propagates the taint information from the DVM layer to the system layer before the native method is executed.

C. System Layer Tracer

To obtain the control flow information at the system layer, the system layer tracer traces each function's entry and exit addresses by hooking each block jump instruction. Note that one function is called when the program counter (PC) jumps to its entry address. When the function exits, PC jumps to its return address.

The system layer tracer collects all native functions' names and their entry addresses. Since all system libraries are in the directory */system/lib/*, we extract all symbols and their offset addresses using *objdump*. These symbols include both the variable symbols and function symbols and we just select function symbols with the symbol marks ('T' stands global method and 't' stands local method). From Android 4.3.1, we extract 56,374 symbols from 204 system libraries. For the third-party functions implemented by apps, we extract them from the app's own libraries in the directory */data/app-lib/*.

When the system layer tracer starts tracing, it gets the beginning addresses of the libraries according to the OS-level semantic knowledge provided by DECAF [25]. The absolute addresses of native functions are calculated according to the offset and their libraries' beginning addresses. The functions' names and absolute addresses are stored in the native function hash table, where hash keys are the functions' absolute addresses and hash values are functions' names. The return address is fetched from the register $R14$ [28]. Since the function calling obeys the nested rule, we use stack structure to store function return addresses.

Information about function entering and exiting is obtained according to the destination address of the block jump instruction. More precisely, we implement a block jump hook routine

to process each block jump instruction before it is executed through the instruction hook interface of DECAF [25].

D. Kernel Layer Tracer

To collect control flow information in the kernel, the kernel layer tracer also needs to first get the kernel functions' entry addresses. All kernel functions and their absolute entry addresses can be parsed from the kernel symbol table of the guest Android system directly. From Android Linux kernel version 3.4, we extract 35728 different kernel symbols.

Kernel symbol table can be read from file *System.map* or *kallsyms*. *System.map* is a real file on the Android system and each kernel has its own *System.map*. *kallsyms* is a "proc file" in the directory */proc/* and it is created on the fly when the Android kernel boots up. Since non-root users cannot extract the addresses of the symbols stored in *kallsyms*, AndroidPerf extracts the kernel symbols and obtains their addresses from *System.map*.

Since tracing functions at the kernel layer is similar to that at the system layer, we just describe how to obtain other important information, such as process switch and exceptions.

There is a special situation that we cannot specify the traced processes' PIDs because the processes' PIDs are not fixed. For example, we cannot know the PID of the process which handles the Ethernet device interrupt and receives the packet arriving at the host. Since the Ethernet device interrupt handling routine starts from function *smc_interrupt()*, we hook this function to acquire the PID of the routine handling the packet receiving interrupt. Note that process switches when the interrupt handling routine finishes. Hence, we can trace the existing of the packet receiving interrupt handling routine through monitoring process switch. Because process context is switched in function *cpu_v7_switch_mm()*, we hook this function to monitor process switch.

TABLE I
OFFSET AND RETURN ADDRESS OF ARM EXCEPTION.

Exception	Offset	Return address
Reset	$0x00$	Not available
Undefined Instruction	$0x04$	ARM: $R14$=PC+4 Thumb: $R14$=PC+2
SWI	$0x08$	ARM: $R14$=PC+4 Thumb: $R14$=PC+2
Abort (data)	$0x10$	$R14$=PC+8
Abort (prefetch)	$0x0c$	$R14$=PC+4
FIQ	$0x18$	$R14$=PC+4
IRQ	$0x1c$	$R14$=PC+4

AndroidPerf also handles ARM exceptions. There are 7 different types of exceptions stored in exception vector table on ARM platform [21]. The vector table actually contains control transfer instructions that jump to the respective exception handlers. And the location of the exception vector addresses is configured through setting the V bit in register $CP15$ [29], where 0 means base address of the vector is $0x00000000$ and 1 means the base address is $0xffff0000$. AndroidPerf first gets the V bit in register $CP15$ to know the exception vector address, and then calculates the absolute addresses of all exception entries according to their offset addresses in Table I. Because exception handlers are not entered and exited through jump instruction, we hook each ARM instruction to trace exception handlers' entries and exits.

Since exceptions also obey the nested rule, AndroidPerf uses stack to store exceptions' return addresses. When the program counter (PC) enters an exception handing routine, AndroidPerf obtains the exception type and calculates the exception's return address according to the exception processing rules in Table I, and then pushes the return addresses into exception stack. When PC branches to the return address on the top of the exception stack, AndroidPerf logs this event and pops its return address from the stack.

E. Taint Propagation Engine

AndroidPerf creates shadow memory tables to save the taint labels of real memory and shadow registers to maintain the taint labels of real registers, and refers to them when the taint information is propagated. To save memory, only the addresses with taint labels will be stored in the shadow memory table. In order to improve the taint lookup efficiency, AndroidPerf uses hash table to implement the shadow memory table, where the memory addresses serve as the hash keys.

AndroidPerf propagates taints according to the taint propagation policy logic. Among 148 ARM and 73 Thumb instructions, we found that 101 ARM and 55 Thumb instructions affect taint propagation after manual analysis. Table II lists the taint propagation logic for general types of ARM/Thumb instructions that can affect taints propagation.

$binary-op$ represents binary operators(e.g., add, sub, etc.); $unary-op$ denotes unary operators(e.g., NOT, etc.); R_d and R_n indicate ARM registers; $\#Imm$ is an immediate number; $M[addr : addr+n]$ denotes memories from $addr$ to $addr+n-1$; | represents the union operation; $Cal(R_n, \#Imm)$ calculates the result based on R_n and $\#Imm$; $t(R_d)$ represents the taint of register R_d; $t(M[addr : addr+n])$ denotes the taints of the memories starting from $addr$ to $addr+n-1$; LDM/STM denote the load/store multiple values instruction from/to memory, $POP/PUSH$ represent the special cases of LDM/STM where R_n = SP, and SWP donate the values switch instruction. For instructions of types $LDR*/STR*$, we set the taint of R_d to the union of $t(M[addr : addr+n])$ and $t(R_n)$, because $addr$ is calculated based on R_n and $\#Imm$. That is, if the tainted input is the address of an untainted value, the taint will be propagated to it.

IV. EVALUATION

In this section, we evaluate AndroidPerf's functionality and overhead. We run the dynamic taint analysis sub-system on a desktop running Ubuntu 14.04 system with Intel i7 CPU and 32G memory. The guest system in the emulator is a modified Android version 4.3 system with Linux kernel 3.4. Moreover, we run the instrumentation sub-system on a Nexus 4 smartphone.

A. Cross-layer Call Graphs

To evaluate AndroidPerf, We develop an app with several operations, including writing data into a file in */sdcard*, sending

TABLE II

TAINT PROPAGATION LOGIC FOR ARM/THUMB/THUMB2 INSTRUCTIONS: SYMBOL " $\circ$ " INDICATES BINARY OPERATORS, SYMBOL " $\sim$ " INDICATES UNARY OPERATORS AND SYMBOL"$\neg$" INDICATES A BITWISE LOGICAL NOT OPERATION; 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	Insn Semantics	Policy type	Taint Propagation	Description
binary-op $R_d, R_n, R_m[, \#Imm]$	$R_d = R_n \circ R_m[\circ \#Imm]$	*set*	$t(R_d) = t(R_n) \mid t(R_m)$	set R_d's taint with union of R_n's and R_m's
binary-op $R_d, R_n[, \#Imm]$	$R_d = R_d \circ R_n[\circ \#Imm]$	*add*	$t(R_d) \mid= t(R_n)$	add R_n's taint to R_d
binary-op R_d, R_n, R_m, R_a	$R_d = R_n \circ R_m \circ R_a$	*set*	$t(R_d) = t(R_n) \mid t(R_m) \mid t(R_a)$	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	$< R_dHi : R_dLo >= R_n \circ R_m$	*set*	$t(R_dHi) = t(R_n) \mid t(R_m)$ $t(R_dLo) = t(R_n) \mid t(R_m)$	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	$R_d =\sim R_m$	*set*	$t(R_d) = t(R_m)$	set R_d's taint with R_m's
mov $R_d, \#Imm$	$R_d = \#Imm$	*clear*	$t(R_d) = TAINT\_CLEAR$	clear R_d's taint
mov R_d, R_m	$R_d = R_m$	*set*	$t(R_d) = t(R_m)$	set R_d's taint with R_m's
mov $R_d, R_m, \#Imm$	$R_d = R_m \circ \#Imm$	*set*	$t(R_d) = t(R_m)$	set R_d's taint with R_m's
mvn $R_d, \#Imm$	$R_d = \neg \#Imm$	*clear*	$t(R_d) = TAINT\_CLEAR$	clear R_d's taint
mvn R_d, R_m	$R_d = \neg R_m$	*set*	$t(R_d) = t(R_m)$	set R_d's taint with R_m's
mvn $R_d, R_m, \#Imm$	$R_d = \neg(R_m \circ \#Imm)$ *cset*		$t(R_d) = t(R_m)$	set R_d's taint with R_m's
LDR* $R_t, R_n, \#Imm$	$addr = Cal(R_n, \#Imm)$ $R_t = M[addr : addr + n]$	*set*	$t(R_t) = t(M[addr : addr + n]) \mid t(R_n)$	set R_t's taint with union of R_n's and $M[addr : addr + n]$'s
LDR* R_t, R_n, R_m	$addr = Cal(R_n, R_m)$ $R_t = M[addr : addr + n]$	*set*	$t(R_t) = t(M[addr : addr + n]) \mid t(R_n) \mid t(R_m)$	set R_t's taint with union of R_n's, R_m's and $M[addr : addr + n]$'s
LDR* $R_t, R_n, R_m, \#Imm$	$addr = Cal(R_n, R_m, \#Imm)$ $R_t = M[addr : addr + n]$	*set*	$t(R_t) = t(M[addr : addr + n]) \mid t(R_n) \mid t(R_m)$	set R_t's taint with union of R_n's, R_m's and $M[addr : addr + n]$'s
LDM(POP) $R_n, regList$	$startAddr = R[n]$ $endAddr = R[n] + f(regList) * 4$ $\{R_i, ..., R_{(i+f(regList)-1}\} = M[startAddr : endAddr]$	*set*	$t(\{R_i, ..., R_{i+f(regList)-1}\}) = t(R_n) \mid t(M[startAddr : endAddr])$	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* $R_t, R_n, \#Imm$	$addr = Cal(R_n, \#Imm)$ $M[addr : addr + n] = R_t$ *set*		$t(M[addr : addr+n]) = t(R_t)$	set $M[addr : addr + n]$'s taints with R_t's
STR* R_t, R_n, R_m	$addr = Cal(R_n, R_m)$ $M[addr : addr + n] = R_t$	*set*	$t(M[addr : addr+n]) = t(R_t)$	set $M[addr : addr + n]$'s taints with R_t's
STR* $R_t, R_n, R_m, \#Imm$	$addr = Cal(R_n, R_m, \#Imm)$ $M[addr : addr + n] = R_t$	*set*	$t(M[addr : addr+n]) = t(R_t)$	set $M[addr : addr + n]$'s taints with R_t's
STM(PUSH) $regList$	$startAddr = R[_{SP}] - f(regList) * 4$ $endAddr = R[_{SP}]$ $M[startAddr : endAddr] = \{Ri, ..., R_{(i+f(regList)-1)}\}$	*set*	$t(M[startAddr : endAddr]) = t(\{R_i, ..., R_{(i+f(regList)-1)}\})$	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
SWP* R_t, R_n, R_m	$R_t = M[R_m]$ $M[R_m] = R_n$	*set*	$t(R_t) = t(R_m) \mid t(M[R_m])$ $t(M[R_m]) = t(R_n)$	set R_t's taint with union of R_m's and $M[R_m]$'s and set $M[R_]$'s with R_n's

UDP packets, and so on. Then we run AndroidPerf to trace these three operations, and construct the cross-layer call graphs using the control flow and data flow information extracted from the tracing logs.

The number of calls (edges) and functions (nodes) that are involved by these operations are summarised in Table III. We can observe that the kernel layer has the largest number of function invocations and unique functions. The call graphs of these three operations (only that of *FileOutputStrem.write()* is shown in Fig.4 because of page limit) show that all these operations will call their corresponding system calls through JNI bridge. Moreover, we can observe that the function call from the system layer to the kernel layer goes through SWI (software interrupt).

B. Case Study 1: Comparing File Writing Functions

There are three Java classes providing functions to write data into a file, including *FileOutputStrem*, *BufferedOutputStream* and *FileWriter*. Although we may notice their performance differences, it is not easy to identify the root causes. We use AndroidPerf to facilitate the in-depth analysis.

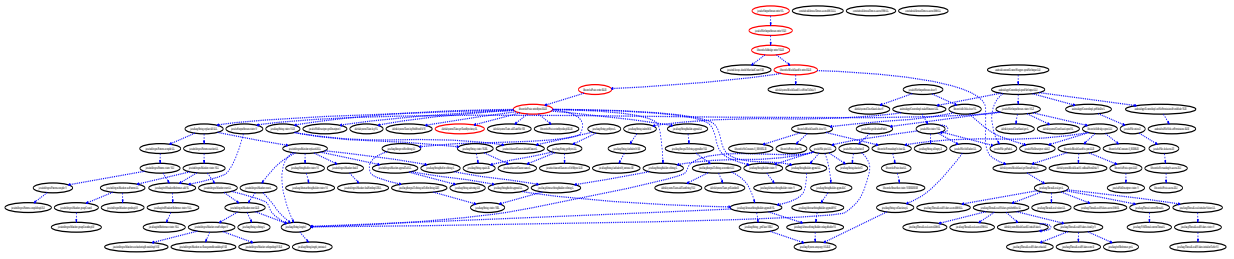
TABLE III

NUMBER OF FUNCTION INVOCATIONS AND THAT OF UNIQUE FUNCTIONS INVOLVED IN THREE OPERATIONS

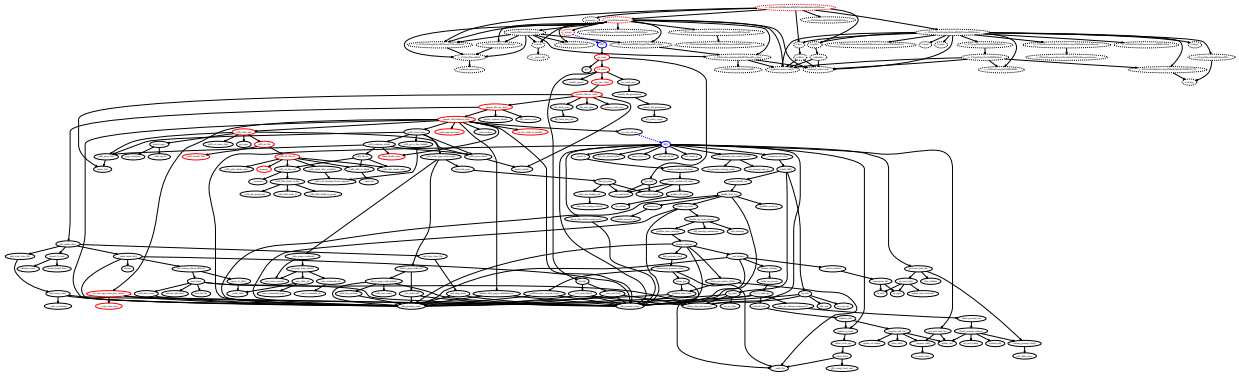
Operations	DVM layer	System layer	Kernel layer	Exception
Socket.connect()	103/70	229/42	2165/388	9
DatagramSocket.send()	169/84	333/46	1337/318	9
FileOutputStream.write()	257/137	190/39	285/170	2

We first developed an app that invokes different functions to store data into files in SDcard. For each test, the app opens one empty file, writes 256 bytes data with different iterations (1000/4000/7000/10000) into this file, then closes this file. The time is calculated from opening file to closing file. For each method, we run 30 times and show the time from opening files to closing files in Fig.5. The result shows that *FileOutputStream* is about five times slower than both *BufferedOutputStream* and *FileWriter*, while *BufferedOutputStream* is only a little bit better than *FileWriter*.

To identify the root causes of the different performance, we run AndroidPerf to trace these three methods and recon-



(a) DVM layer.



(b) System layer & kernel layer.

Fig. 4. Call graph of *FileOutputStream.write()*.

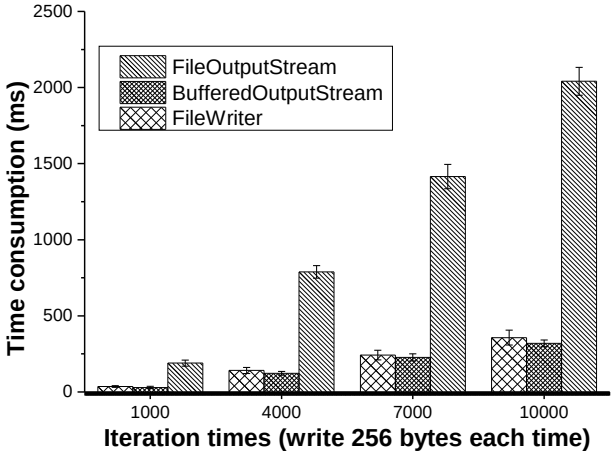
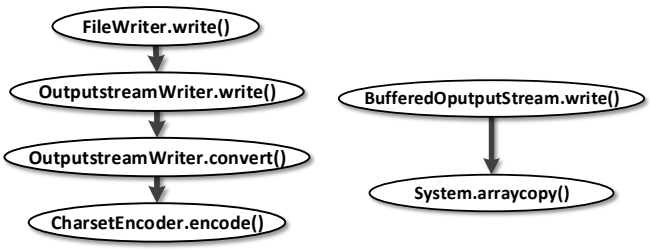


Fig. 5. Comparison between different file writing functions.



(a) *FileWriter.write()* (b) *BufferedOutputStrem.write()*

Fig. 6. Data flow of two file writing functions.

struct their call graphs. The data flows of *FileWriter.write()* and *BufferedOutputStrem.write()* are shown in Fig.6(a) and Fig.6(b) with only key functions, respectively. The control flow and data flow of *FileOutputStream.write()* are showed in Fig.4. It shows that *FileOutputStream* invokes system call *write()* through JNI bridge method *writeBytes()* in *libcore/io/-Posix.java* to write data into the file directly each time.

The results of AndroiPerf help us learn that *FileWriter* and *BufferedOutputStream* have data buffers and just store data in their buffers when the buffers are not full. Therefore, *FileWriter* and *BufferedOutputStream* have much higher efficiency than *FileOutputStream*. We also found that *FileWriter* needs to encode the data when the data is stored in buffer, while *BufferedOutputStream* holds a buffer of arbitrary binary data. Hence, the performance of *FileWriter* is a little bit worse than that of *BufferedOutputStream*.

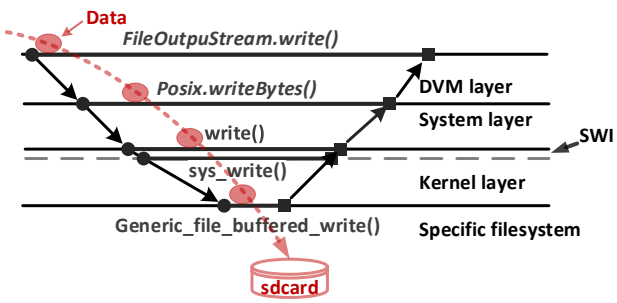


Fig. 7. Major functions involved in file writing at different layers.

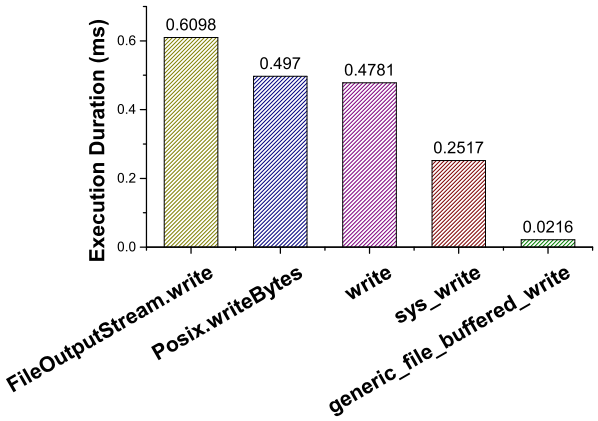


Fig. 8. The time consumptions of major functions involved in file writing at different layers.

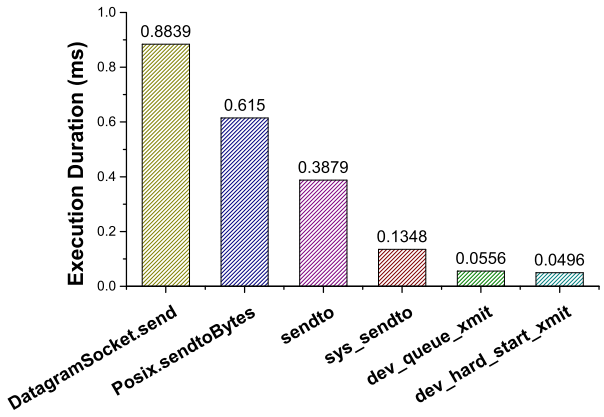


Fig. 10. The time consumptions of major functions involved in UDP packet transmission at different layers.

To evaluate the time consumption of functions on different layers, we capture the execution time of functions at difference layers. The boundaries functions at each layer and their relationship is shown in Fig. 7. Since the function *generic_file_buffered_wite()* allocates page cache and copies data to the page cache, we take its execution time as the time consumption of the filesystem. In this experiment, the app writes 256 bytes data for 30 times and then we calculates the mean execution time of each function. As shown in Fig.8, the major time is consumed by functions at the DVM layer and the system layer.

C. Case Study 2: Analyzing Packet Transmission Procedure

When studying popular network measurement tools, we find that these tools can be divided into two major classes: those working on the system layer (such as *iperf* [30], *Network Tools* [31], *OneProbe* [32], etc.), and those working on the DVM layer(such as *speedtest* [33], *Internet Speed Test* [34], etc.). Only a few tools working in the kernel layer [35], [36]. To shed a light on the performance of different methods, we use *DatagramSocket* to send a UDP packet from the DVM and trace its data flow to find out the entries and exits of each layer. The key functions traced are shown in Fig.9.

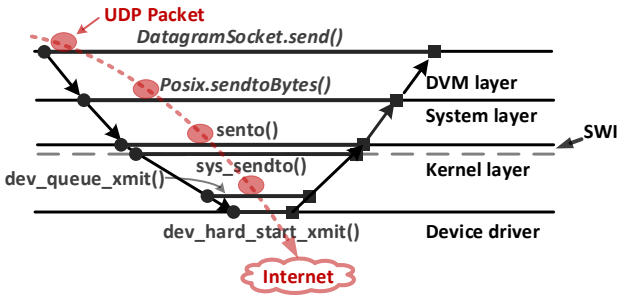


Fig. 9. Major functions involved in UDP packet transmission at different layers.

Fig.10 illustrates the time consumption of each function. We can see that more than the majority of the time are consumed by DVM functions and native functions when a UDP packet is sent out from DVM. The result suggests that when developing smartphone-based measurement tools it would be better to realize them in the kernel layer, such as using the toolkit *kTRxer* [35], to avoid noise introduced by DVM and even the native layer.

D. Overhead

In order to evaluate the overhead brought by AndroidPerf (especially the dynamic taint analysis sub-system), we ran benchmark tool *Mobibench* [37], which executes SQLite operations, on AndroidPerf and unmodified *Qemu*, respectively. As shown in Table IV, AndroidPerf incurs about 8 times slow down when compared with unmodified *Qemu* on SQLite operations, including 8 times slowdown on Insert, 9 times slowdown on Update, and 5 times slowdown on Delete.

We also ran CF-Bench [38] on AndroidPerf and *Qemu*, individually, and illustrate the results in Fig.11. Note that the results of AndroidPerf are calculated according to the performance of *Qemu*, which is taken as the benchmark. In general, AndroidPerf can only achieve 20% of *Qemu*'s performance. Moreover, native operations cause much more slowdown than Java operations. The reason is that AndroidPerf uses extended TaintDroid to perform taint analysis on Java operations executed by DVM whereas the taint analysis on the native layer and the kernel layer is conducted in *Qemu* at the instruction level. Note that TaintDroid just introduces 14% overhead with respect to the unmodified Android system [15] while the instruction level tracer, such as the taint tracker in DroidSope [16], can incur more than 11 times slowdown. Fortunately, since AndroidPerf does not perform all taint analysis at the instruction level, it will have better efficiency than the taint tracker in DroidSope. Moreover, AndroidPerf only outputs the collected information to log files after the execution of certain operation or an app finishes and then analyzes the log files, thus further reducing the overhead of tracing information flows.

V. Discussion

The major goal of AndroidPerf is to track function calls and conduct dynamic taint analysis in the DVM layer, system layer, and kernel layer. It also modifies Android's profiling

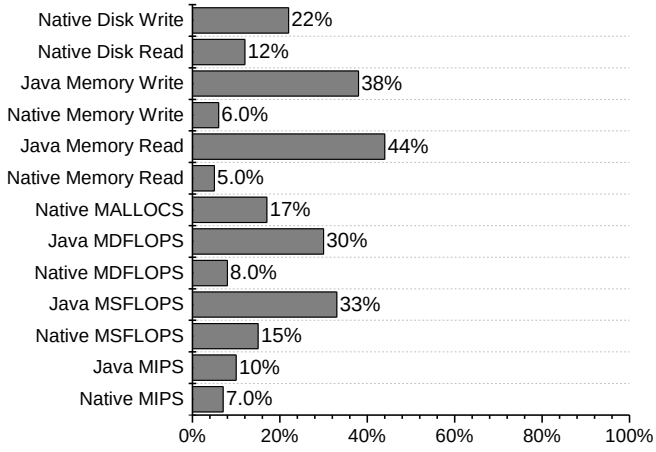


Fig. 11. CF Benchmark results.

TABLE IV
OVERLOAD COMPARISON BETWEEN ANDROIDPERF AND *Qemu* ON SQLITE OPERATIONS

Operation	SQLite.Insert	SQLite.Update	SQLite.Delete
AndroidPerf	22 TPS	29 TPS	48 TPS
QEMU	176 TPS	252 TPS	193 TPS

framework and adopts dynamic instrumentation to collect performance data from different layers. Although selecting functions for different purposes is out of the scope of this paper, we will investigate approaches like SIF [39] to facilitate users to select critical functions.

Since how to automatically drive apps to cover all paths is still an open question, we will integrate AndroidPerf with new testing systems like A^3E [40] and Dynodroid [41] for increasing the path coverage in future work.

VMI-based approaches have the same limitation of emulating a real hardware environment [23]. In other words, the emulator may miss some important information sources, especially those from specific hardware. Moreover, apps may differentiate between an emulator and a real smartphone by exploiting their difference. We will explore virtualization technology, such as Trustzone in ARM [42], to run the whole system of AndroidPerf in a real smartphone.

AndroidPerf has not been tested on the new Android runtime, ART, which became the default runtime since Android 5.0 [43]. We believe AndroidPerf can be extended to support ART and will do it in future work, because apps will be converted to native codes before execution by ART and AndroidPerf can handle native codes.

VI. RELATED WORK

A. Profiling Android Applications

A number of systems have been designed to profile Android apps for different purposes, such as identifying performance issues [11], [44], detecting malware [15], [17], modeling energy consumption [7], [8], etc. However, none of them achieves the same functionality as AndroidPerf.

Google offers Traceview and dmtracedump to trace invoked functions and collect time spent in each function [44]. Although the trace logs can be generated by including the *Debug* class in an app or using DDMS, Google recommends the former to get more precise results [44]. In contrast, AndroidPerf can trace function calls and conduct dynamic taint analysis *without* modifying apps.

While some systems instrument apps and/or the system to locate performance issues, they neither trace invoked functions nor collect information from the system and the kernel, and thus cannot reveal issues due to the underlying platform [5], [6], [9], [10]. For example, to measure user-perceived transaction, Panappticon instruments event handlers, asynchronous call interfaces, and the interprocess communication mechanisms to log events [9]. Similarly, AppInsight instruments apps for Windows Mobile to identify the critical execution path for locating performance bottlenecks [5], [6], [10].

Some systems collect information from different sources to profile apps or model energy consumptions [7], [8], [11]–[14]. However, they neither track all function calls at different layers nor perform the dynamic taint analysis, and thus cannot uncover issues due to the poor interactions between different layers. For example, ProfileDroid collects user-generated events through adb, system calls through strace, and network traffic through tcpdump to profile apps [11]. ARO characterizes resource usage by correlating user input events and network traffic [12]. QoE Doctor further correlates user interaction events, network traffic, and RRC/RLC layer data through QxDM to diagnose apps QoE [13]. To estimate energy consumption, AppScope monitors an app's hardware usage by probing system calls relevant to hardware operations at the kernel level [8], [14]. Being a fine-grained energy profiler, Eprof collects DVM level function calls and system calls by modifying Android framework [7]. If an app has native components, Eprof requires linking it with the Android gprof library [7]. Note that AndroidPerf can traces more functions than system calls and differentiate them through taint analysis.

Although VARI profiler can trace invoked functions in the DVM layer, system layer, and kernel layer, it does not perform dynamic taint analysis and thus cannot track information flow across different layers [45].

B. Dynamic Taint Analysis

Dynamic taint analysis [20] has been adopted to analyze apps, such as detecting private information leakage [15], [18], dissecting malware [16], [17], etc., and a few systems have been released, such as TaintDroid [15], DroidScope [16], and NDroid [18]. However, none of them can conduct the cross-layer taint analysis like AndroidPerf. More precisely, TaintDroid modifies DVM to track information flows within DVM [15]. Some systems integrated TaintDroid with other functionality, such as tracing APIs and system calls [17], [46]. Although DroidScope does not release its taint tracker, it provides a great framework to build dynamic taint analysis tools, because it can reconstruct detailed information in Linux and DVM through VMI [16]. NDroid, built on top of DroidScope,

can track information flows between the DVM layer and the native layer [18]. However, NDroid only considers third-party native libraries. Note that AndroidPerf conducts dynamic taint analysis on *all* functions at different layers.

VII. Conclusion

To reveal hidden issues in apps due to the underlying platform or poor interactions between different layers, it is desirable to profile apps from all layers, including the DVM layer covering apps and the Android framework, the system layer containing system or third-party native libraries, and the kernel layer comprising exported system calls and internal functions. In this paper, we propose AndroidPerf, the first cross-layer profiling system for Android apps. AndroidPerf consists of a sub-system to perform cross-layer dynamic taint analysis and a sub-system to conduct instrumentation on different layers. After tackling a number challenging issues, we have realized AndroidPerf in 9,125 lines of C/C++ and 1,016 lines of Python scripts along with some modifications to Android's framework. The evaluation results through real case studies demonstrate AndroidPerf's effectiveness and efficiency.

VIII. Acknowledgment

We thank the anonymous reviewers for their quality reviews and suggestions. This work is supported in part by the Hong Kong GRF (No. PolyU 5389/13E), the National Natural Science Foundation of China (No. 61202396), the PolyU Research Grant (G-UA3X), the Open Fund of Key Lab of Digital Signal and Image Processing of Guangdong Province (2013GDDSIPL-04), and the Shenzhen City Special Fund for Strategic Emerging Industries (No. JCYJ20120830153030584).

References

[1] I. Corporate, "Android and ios squeeze the competition, swelling to 96% of the smartphone operating system market for both 4q14 and cy14," http://goo.gl/8fFSu1, Feb. 2015.
[2] AppBrain, "Number of android applications," http://goo.gl/TOX992, Mar. 2015.
[3] H. Zhu, H. Xiong, Y. Ge, and E. Chen, "Mobile app recommendations with security and privacy awareness," in *Proc. KDD*, 2014.
[4] C. Qian, X. Luo, L. Yu, and G. Gu, "Vulhunter: toward discovering vulnerabilities in android applications," *IEEE Micro*, vol. 35, no. 1, 2015.
[5] L. Ravindranath, J. Padhye, S. Agarwal, R. Mahajan, I. Obermiller, and S. Shayandeh, "Appinsight: Mobile app performance monitoring in the wild." in *Proc. OSDI*, 2012.
[6] L. Ravindranath, J. Padhye, R. Mahajan, and H. Balakrishnan, "Timecard: Controlling user-perceived delays in server-based mobile applications," in *Proc. SOSP*, 2013.
[7] A. Pathak, Y. C. Hu, and M. Zhang, "Where is the energy spent inside my app?: Fine grained energy accounting on smartphones with eprof," in *Proc. EuroSys*, 2012.
[8] C. Yoon, D. Kim, W. Jung, C. Kang, and H. Cha, "Appscope: application energy metering framework for android smartphones using kernel activity monitoring," in *Proc. USENIX ATC*, 2012.
[9] L. Zhang, D. Bild, R. Dick, Z. Mao, and P. Dinda, "Panappticon: event-based tracing to measure mobile application and platform performance," in *Proc. CODES+ISSS*, 2013.
[10] L. Ravindranath, S. Nath, J. Padhye, and H. Balakrishnan, "Automatic and scalable fault detection for mobile applications," in *Proc. MobiSys*, 2014.
[11] X. Wei, L. Gomez, I. Neamtiu, and M. Faloutsos, "Profiledroid: Multi-layer profiling of android applications," in *Proc. MobiCom*, 2012.
[12] F. Qian, Z. Wang, A. Gerber, Z. Mao, S. Sen, and O. Spatscheck, "Profiling resource usage for mobile applications: a cross-layer approach," in *Proc. ACM Mobisys*, 2011.
[13] Q. Chen, H. Luo, S. Rosen, Z. Mao, K. Iyer, J. Hui, K. Sontineni, and K. Lau, "Qoe doctor: Diagnosing mobile app qoe with automated ui control and cross-layer analysis," in *Proc. ACM IMC*, 2014.
[14] S. Lee, C. Yoon, and H. Cha, "User interaction-based profiling system for android application tuning," in *Proc. Ubicomp*, 2014.
[15] W. Enck, P. Gilbert, B. Chun, L. Cox, J. Jung, P. McDaniel, and A. Sheth, "Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones," in *Proc. OSDI*, 2010.
[16] L. Yan and H. Yin, "Droidscope: Seamlessly reconstructing OS and Dalvik semantic views for dynamic Android malware analysis," in *Proc. USENIX Security*, 2012.
[17] V. Rastogi, Y. Chen, and W. Enck, "Appsplayground: Automatic large-scale dynamic analysis of android applications," in *Proc. CODASPY*, 2013.
[18] C. Qian, X. Luo, Y. Shao, and A. T. Chan, "On tracking information flows through jni in android applications," in *Proc. DSN*, 2014.
[19] S. Ratabouil, *Android NDK Beginner's Guide*. Packt Publishing, 2012.
[20] E. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in *Proc. IEEE SP*, 2010.
[21] "Arm exceptions," http://www.ethernut.de/en/documents/arm-exceptions.html, 2009.
[22] F. Bellard, "QEMU, a fast and portable dynamic translator," in *Proc. USENIX ATC, FREENIX Track*, 2005.
[23] T. Garfinkel and M. Rosenblum, "A virtual machine introspection based architecture for intrusion detection," in *Proc. NDSS*, 2003.
[24] S. A. Siegfried Rasthofer and E. Bodden, "A machine-learning approach for classifying and categorizing android sources and sinks," in *Proc. NDSS*, 2014.
[25] A. Henderson, A. Prakash, L. K. Yan, X. Hu, X. Wang, R. Zhou, and H. Yin, "make it work, make it right, make it fast building a platform-neutral whole-system binary analysis platform," in *Proc. ACM ISSTA*, 2012.
[26] "strace," http://sourceforge.net/projects/strace.
[27] J. Keniston, P. S. Panchamukhi, and M. Hiramatsu, "Kernel probes (kprobes)," https://www.kernel.org/doc/Documentation/kprobes.txt.
[28] "Processor core register summary," http://goo.gl/Cem8bp.
[29] "Exception vectors," http://goo.gl/1MMK1z.
[30] "iperf for android," http://goo.gl/BcTP9U.
[31] "Network tools," http://goo.gl/7qdq1e.
[32] X. Luo, E. Chan, and R. Chang, "Design and implementation of TCP data probes for reliable and metric-rich network path monitoring," in *Proc. USENIX Annual Tech. Conf.*, 2009.
[33] "Speedtest.net," http://goo.gl/7ZyoY2.
[34] "Internet speed test," http://goo.gl/dlVRNV.
[35] L. Xue, X. Luo, and Y. Shao, "ktrxer: A portable toolkit for reliable internet probing," in *Proc. IEEE IWQoS*, 2014.
[36] D. Turull, "pktgen," http://people.kth.se/ danielt/pktgen/.
[37] S. Jeong, K. Lee, J. Hwang, S. Lee, and Y. Won, "Androstep: Android storage performance analysis tool." in *Proc. European Workshop on Mobile Engineering*, 2013.
[38] "Cf-bench," http://bench.chainfire.eu/, 2013.
[39] S. Hao, D. Li, W. G. Halfond, and R. Govindan, "Sif: A selective instrumentation framework for mobile applications," in *Proc. MobiSys*, 2013.
[40] T. Azim and I. Neamtiu, "Targeted and depth-first exploration for systematic testing of android apps," in *Proc. OOPSLA*, 2013.
[41] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: An input generation system for android apps," in *Proc. ESEC/FSE*, 2013.
[42] ARM Ltd., "Trustzone," http://goo.gl/AjRTgD.
[43] A. Frumusanu, "A closer look at android runtime (art) in android l."
[44] "Profiling with traceview and dmtracedump," http://goo.gl/QZRXEW.
[45] T. H. Su, H. J. Tsai, K. H. Yang, P. C. Chang, T. F. Chen, and Y. T. Zhao, "Reconfigurable vertical profiling framework for the android runtime system," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 13, 2014.
[46] "Droidbox," https://code.google.com/p/droidbox/.