TaintART: A Practical Multi-level Information-Flow Tracking System for Android RunTime

Mingshen Sun
The Chinese University of Hong Kong

Tao Wei
Baidu X-Lab

John C.S. Lui
The Chinese University of Hong Kong

ABSTRACT

Mobile operating systems like Android failed to provide sufficient protection on personal data, and privacy leakage becomes a major concern. To understand the security risks and privacy leakage, analysts have to carry out data-flow analysis. In 2014, Android upgraded with a fundamentally new design known as Android RunTime (ART) environment in Android 5.0. ART adopts ahead-of-time compilation strategy and replaces previous virtual-machine-based Dalvik. Unfortunately, many data-flow analysis systems like TaintDroid [19] were designed for the legacy Dalvik environment. This makes data-flow analysis of new apps and malware infeasible. We design a multi-level information-flow tracking system for the new Android system called TaintART. TaintART employs a multi-level taint analysis technique to minimize the taint tag storage. Therefore, taint tags can be stored in processor registers to provide efficient taint propagation operations. We also customize the ART compiler to maximize performance gains of the ahead-of-time compilation optimizations. Based on the general design of TaintART, we also implement a multi-level privacy enforcement to prevent sensitive data leakage. We demonstrate that TaintART only incurs less than 15 % overheads on a CPU-bound microbenchmark and negligible overhead on built-in or third-party applications. Compared to legacy Dalvik environment in Android 4.4, TaintART achieves about 99.7 % faster performance for Java runtime benchmark.

1. INTRODUCTION

Mobile devices such as smartphones, tablets and wearable devices are widely used for communication, photo taking, entertainment, and monitoring health status. Many applications (apps for short) installed on the smartphones provide useful services, but they may also privately send sensitive information to remote servers for various data analytics [12]. Worse yet, some of them can gain profit from these personal data [38]. Furthermore, malware can secretly steal sensitive information such as contact lists without users' consent. All these indicate that privacy leakage is a serious threat to a large community of mobile users.

To understand the possibility of privacy leakage, researchers seek solutions in two directions of data-flow analysis. Firstly, with the disassembled code of a given app, researchers can perform static data-flow analysis techniques such as static taint analysis and symbolic execution. This type of methods can statically derive a set of possible data which may leave devices at runtime, and decide whether sensitive data leaks to untrusted channels. The limitation of this method is that it cannot detect runtime information disclosures when the app developers use techniques such as code with Java reflection, code encryption, or dynamic code loading techniques. Therefore, researchers proposed to use dynamic methodologies to monitor suspicious behaviors at runtime. The dynamic taint analysis technique [46] is one of many dynamic methodologies which can track the information flows within apps at runtime. The dynamic taint analysis technique will label (*taint*) sensitive data from certain sources and handle label transitions (*taint propagation*) between variables, files, and procedures at runtime. If a tainted label transmits out of the mobile device through some functions (sinks), one can then monitor the data leakage dynamically. This method can accurately track data flows at an app's execution time.

TaintDroid [19] is a notable dynamic taint analysis system for Android apps. It customizes Android runtime (Dalvik Virtual Machine) to achieve taint storage and taint propagation. Many systems [16, 63, 5, 43, 42, 54] are based on TaintDroid to conduct further analysis. However, there are several constraints which make TaintDroid can no longer function on the latest Android for privacy tracking and malware analysis (and to a certain extent, data flow analysis).

Firstly, TaintDroid was originally designed for virtual-machine-based system (i.e., Dalvik virtual machine), and implemented on legacy Android systems 2.1, 2.3, 4.1, and 4.3. TaintDroid utilizes the internal memory of Dalvik virtual machine for taint storage and propagation. To enhance the performance of Android, Google recently changed to the ahead-of-time (AOT) compilation strategy and introduced Android RunTime (ART) to replace Dalvik VM starting from Android 5 and onward. Instead of interpreting code (or using JIT [25, 62]) by virtual machine at runtime, the AOT compilation strategy will directly compile apps into native code at the first installation time. Therefore, one cannot use TaintDroid for the newly-designed runtime and

CCS'16, October 24 - 28, 2016, Vienna, Austria

ISBN 978-1-4503-4139-4/16/10...$15.00

DOI: http://dx.doi.org/10.1145/2976749.2978343

TaintDroid can at most only support legacy systems up to Android 4.4.

Secondly, although the latest Android still provides a fallback runtime interpreter for debugging, the performance is not acceptable (as shown in our evaluation). Therefore, porting TaintDroid to this fallback runtime cannot take advantage of compiler optimization and the performance issue hinders effective security and data flow analysis.

Thirdly, because of the compatibility and performance issues, users cannot use TaintDroid for policy enforcement to prevent privacy leakage. As shown in the Android distribution statistics [24], about half of Android users have already upgraded to Android 5.0 or above, and this number continues to grow.

Last but not least, we discover that app developers tend to target newer Android versions so as to use latest features which TaintDroid does not support. We measured SDK versions of five hundreds apps in Google Play's "Top Charts" on October 2015 and February 2016. As shown in Figure 1, the average target SDK version has changed from 19 (Android 4.4) to 20 (Android 5.0). Again, this implies that many new apps may not be analyzed by TaintDroid, and malware can exploit this incapability to bypass security or data flow analysis.

In this paper, we design and implement TaintART, a dynamic information-flow tracking system which targets the latest Android runtime. TaintART introduces a multi-level taint label so as to tag the sensitive levels of different taint sources. TaintART instruments Android's compiler and utilizes processor registers for taint storage. Compared to TaintDroid which needs at least two memory accesses, TaintART only needs few register accesses and hence achieves faster taint propagation. Therefore, TaintART has a much better performance than TaintDroid. We also prototype TaintART on the latest Android system and conduct extensive performance evaluation.

In summary, we make the following contributions:

- **Methodology.** We first propose a novel method to efficiently track dynamic information flows on the Android mobile operating system with ahead-of-time compilation strategy. By instrumenting the compiler, we conduct multi-level taint analysis on compiled apps with optimized code rather than the app's original bytecode. Furthermore, instead of relying on memory storage, our method utilizes processor registers to achieve fast taint storage and propagation, resulting in minimal performance and memory overheads.

- **Implementation.** We implement TaintART on the latest released Android system (Android 6.0 "Marshmallow") which supports the newly-designed application runtime (i.e., ART runtime). TaintART can track multi-level information flows within a method, across methods, as well as data transmitted between apps. To the best of our knowledge, this is the first information-flow tracking system which supports the newly-designed ART runtime. Furthermore, because information-flow analysis is a general analysis technique which can be used in many research areas, we shall open-source our system.

- **Performance.** We also extensively perform the macrobenchmarks, microbenchmarks and compatibility test of TaintART. TaintART incurs an overall Java runtime overhead of less than 15 % compared to the original environment with optimizing compiler backend. It is worth noting that TaintART can achieve 2.5 % and 99.7 % *faster* for overall performance compared to quick compiler backend ART runtime and Dalvik VM in Android 4.4. In addition, TaintART incurs negligible memory overhead and less than 5 % IPC overhead. More importantly, our CTS test shows that TaintART can analyze apps without compatibility issues.

- **Application to privacy leakage analysis.** On top of TaintART platform, we discover privacy leakage issues on popular apps in Android 6.0 and provide a solution to prevent data leakage in various levels. Furthermore, we also find that some functions of apps could not be analyzed due to compatibility issues when we analyze these apps using TaintDroid based on Android's legacy runtime.

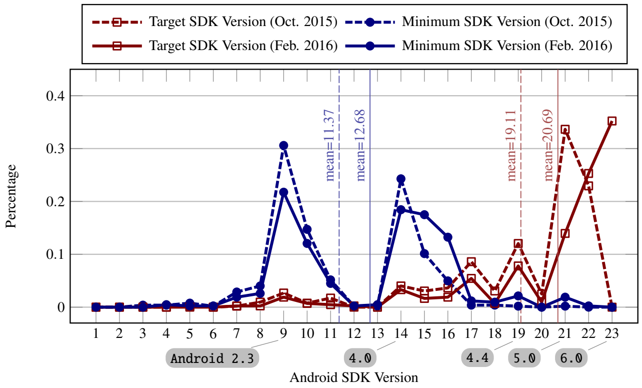


Figure 1: Trends of minimum SDK versions and target SDK versions for apps downloaded from Google Play's "Top Charts".

The rest of this paper is organized as follows. Section 2 introduces the background of Android runtime. Section 3 describes our TaintART design on taint storage, taint propagation, and taint logic. In Section 4, we show the implementation details of TaintART. We also conduct several case studies such as privacy tracking in Section 5. In Section 6, we extensively evaluate the macrobenchmarks, microbenchmarks and compatibility issues of TaintART. Section 7 presents TaintART's limitations and our future works. Related work is presented in Section 8 and Section 9 concludes.

2. BACKGROUND

In this section, we discuss essential background of Android systems and Android app environment.

2.1 Android Overview

Android operating system is based on the Linux kernel. On top of the kernel, Android provides a set of libraries, such as database libraries and app runtime libraries. Moreover, there is a middleware called application framework based on these common libraries. The application framework provides various APIs for apps developers, such as activity management, content management, and view system. Supported by app frameworks, many apps and background services run on the device. There are some system services providing fundamental functions such as sending and receiving messages, getting current locations, and reading accelerometer data. Android apps are mainly written in Java, but to enhance performance, developers can also embed C/C++ and use Java Native Interface (JNI) to interact with apps and framework APIs. Each app runs in an isolated environment. Apps can communicate with other apps and services through a specific inter-process communication mechanism called the binder. Messages in the binder can hold actions or data object references and will be serialized into *parcels*. A binder kernel module is responsible for passing parcel messages between processes. Using this approach, different apps can request actions or share information across app sandboxes.

2.2 Android App Environment

After developing an app, developers need to compile Java sources of an app into bytecode by a Java compiler with the `javac` tool and convert it to the dex bytecode with the `dx` tool. With other resources, the `dex` file will be zipped into one single application package file (`apk` file) for distribution on app markets. Users can download and install apps from the markets into their Android devices. Basically, Android app environment contains two stages: installation stage and runtime stage. During the installation of apps, Android devices may conduct further compilation or optimizations locally. After installation, apps can run with the support of app runtime such as handling method calls to framework APIs, or interacting with the Java native interface (JNI). For Android devices, there are two fundamentally different app environments: Dalvik and ART. Figure 2 illustrates the basic flow of app environment and compares differences of these two environments.

Dalvik Environment Legacy Android systems (versions which are less than 4.4) are equipped with Dalvik environment. Dalvik adopts virtual machine interpretation strategy at runtime. During the app installation, a `dexopt` tool will optimize original dex bytecode such as pre-computing data, pruning empty methods, and improving virtual method calls. At runtime, a Dalvik virtual machine will interpret bytecode and execute architecture specific native code. Dalvik VM maintains an internal stack for local variables and arguments. To improve performance, Dalvik also features modular interpretation and just-in-time compilation.

The modular interpretation technique is to split each opcode in platform-specific modules. For example, the `add-int` dex operation will be interpreted as ARM assembly in the `OP_ADD_INT_LIT16.S` file. TaintDroid modified related modules in the Dalvik VM to implement taint tracking functions.

ART Environment ART was first introduced as an experimental environment along with Android 4.4 in 2014. Starting from Android 5.0 in 2015, Google decided to replace Dalvik and made ART as the default environment. To improve the runtime performance, ART adopts ahead-of-time (AOT) compilation strategy instead of virtual machine interpretation. ART provides a compiler called `dex2oat`. The `dex2oat` tool will directly compile dex bytecode into native code during app's installation and then store as an `oat` file. Because of the AOT compilation, the `dex2oat` compiler can perform multiple passes for optimization to achieve better performance. For historical reasons, there are two compiler backends in the ART compiler which are "quick" backend and "optimizing" backend. The "optimizing" backend can perform more optimization strategies, and became the default compiler backend from Android 6.0. At runtime, ART will mainly handle dynamic heap management such as object allocations and garbage collections. Note that although the original sources of Dalvik VM were removed from the code bases, some names of tools and functions still contains "dalvik".

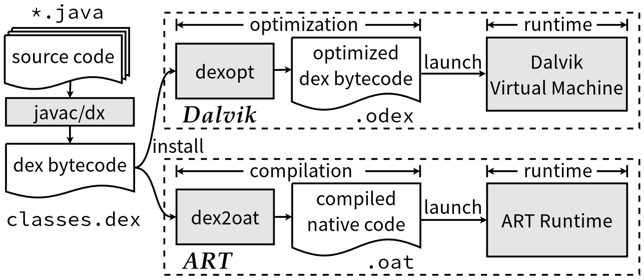


Figure 2: Comparison between Android Dalvik and ART environment.

Because of the differences of the modular interpreter and AOT compiler, the design methodologies of taint tracking are fundamentally different. In addition, TaintDroid customizes Dalvik VM and utilizes its internal stack. It doubles the size of stack frame and stores taint tags for each parameters and local variables in the extra memory slots. This incurs two times internal memory usage for stack frame and at least two memory accesses for each taint propagation event. To better utilize compiler optimization, we propose to instrument the ART compiler.

3. SYSTEM DESIGN

In this section, we first present an overview of TaintART and then discuss various building blocks of our taint-based dynamic information-flow tracking system.

3.1 Overview

Design We design TaintART, a compiler-instrumented information-flow tracking system. TaintART utilizes dynamic taint analysis technique and can track data by inserting tracking logic. TaintART employs a multi-level taint tag methodology to minimize taint storage so that tags can be stored in processor registers for fast access. We implement TaintART by customizing the ART compiler to retain the original ahead-of-time optimizations (which will be presented in Section 3.2). TaintART also defines multi-level data tracking strategy which can be used for policy enforcement on data leakage. Because the compiler and calling convention are stable across versions, TaintART is durable and can be easily updated to support future versions.

In dynamic taint analysis, sensitive data is targeted at any sensitive function called *taint source*. A *taint tag* will be labeled on the sensitive data for tracking. When the data is copied or transformed to another place, its taint tag will *propagate* to the new place. When the data is purged, its taint tag will be *cleared*. We call taint propagation and taint purging as *taint logic* and it defines the transition of taint tag. The taint tag status for tracking data will be stored in *taint tag storage*. Dynamic taint analysis will track the tainted sensitive data and monitor if any tainted data leaves the system at some specified functions called *taint sinks*, such as sending out data via the network or save the data in an external storage.

Figure 3 describes the overview of TAINTART, which consists of two components taking actions on two separate stages respectively. They are TAINTART compiler at installation stage and TAINTART runtime at runtime stage. For the installation stage, the TAINTART compiler will compile apps into native code. Note that the compiler is based on the ART compiler with the "optimizing" backend containing three basic building blocks: builder, optimizer, and code generator. The builder will parse app's dex bytecode to intermediate representations, i.e., internal control flow graphs. Using this internal control graph, the optimizer will combine logic, optimize register assignment, eliminate instructions, etc. Finally, the code generator will compile internal representations into machine specific native code. Before generating the native code, the TAINTART compiler will insert code blocks to handle taint logic. For example, if a tainted variable is copied to another variable, the inserted block will help to propagate taint tags of these variables and modify tag status in the taint storage. Note that code blocks will be injected in a fully optimized code, and this will not only maintain the original program logic, but also preserve performance gains by compiler's optimizations. For the runtime stage, the TAINTART runtime can track taint tag of sensitive data by efficiently accessing tag status in the taint tag storage. When the tainted data is transported to other channels, the TAINTART runtime can report the event.

Figure 3 illustrates a simple control flow graph. Each node in the graph represents a program logic and arrows pointing to the next logic. Node 1 contains an instruction to get sensitive data from a taint source and save in the `R0` variable. Node 2 is to empty `R0` and its taint tag will be clear. Node 3 is to assign the data in `R0` to `R1`. This means the taint tag in `R0` will propagate to `R1`. For taint propagation logic, we will insert nodes (e.g., nodes highlighted in gray) to manage changes of taint tags. For node 4, the logic is to send the data in `R1` to other untrusted channels such as the WiFi network, i.e., a taint sink. As shown in the figure, there are two possible paths at runtime resulting in different data-flow. Only the first one (i.e., red path in solid line) leads to information leakage. TAINTART will track the taint status of each registers (i.e., variables) so as to determine if the data is leaked. If the runtime control flow is the blue path (in dotted line), the taint tag of `R1` should be false in the end meaning no sensitive data is leaked. We will explain how TAINTART efficiently stores taint tags and handles taint logic in later sections.

3.2 Taint Tag Storage

At runtime, each taint logic may cause status change of taint tags. Therefore, the design of taint tag storage will largely affect the runtime performance. TAINTART employs *processor registers* for taint tag storage to achieve the fastest storage access. To illustrate, we consider the scenario for recording two taint tag states only. Essentially, TAINTART can utilize m bits of a register to store a taint tag status of a variable. If a variable is tainted, its corresponding bits in the taint storage register will be marked as tainted, otherwise, the bits will be set to untainted. We first use one bit to represent two taint states, and will discuss the scenario on multiple taint tag states in Section 4.

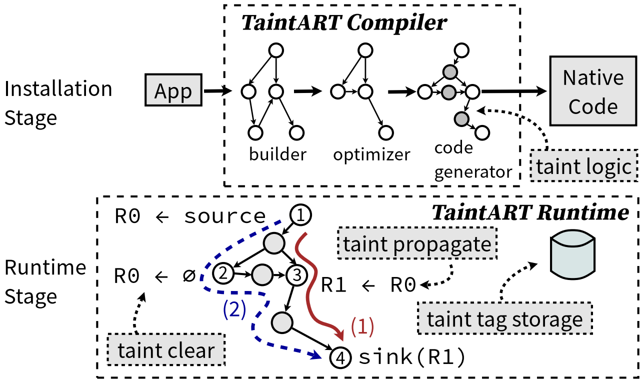


Figure 3: Overview of TaintART.

	S15	…	S4	S3	S2	S1	S0	R15	…	R4	R3	R2	R1	R0
R5	0	…	0	0	0	1	1	0	…	1	0	0	1	0

floating-point registers regular registers

Figure 4: Taint tag storage using register `R5`.

Specifically, the TAINTART compiler will reserve registers for taint storage. Figure 4 illustrates the basic idea of storing taint tag in registers. Our TAINTART prototype is built on Google Nexus 5, which is a 32-bit ARM platform Android device. There are 16 regular CPU registers and each register has 32 bits. We reserve the register `R5` for taint storage. The register allocator of the TAINTART compiler will ensure that `R5` will not be assigned for other purposes such as variable storage. The first sixteen bits (from bit 0 to bit 15) will be used for storing taint tags of sixteen registers (from `R0` to `R15`). Note that the ART runtime will also reserve stack registers (`SP/R13`, `LR/R14` and `PC/R15`), thread register (`TR/R9`) and temp register (`R12`). Therefore, we do not need to maintain bits for taint tag storage of these registers. Besides, Nexus 5 contains a vector floating-point coprocessor. We use the remaining sixteen bits for storing taint tag of floating point registers (from `S0` to `S15`).

Taint Tag Spilling Because a processor has a limited number of registers and not all variables can be assigned to registers, the register allocator will temporarily store extra variables in the main memory. The operation of moving a variable from a register to the main memory is called *register spilling*. The taint tag storage of TAINTART is based on CPU registers. If a variable in register is spilled to memory, the taint tag of this register is no longer valid. In our design, we will store its taint tag into memory right after the spilled variable. We call this operation *taint tag spilling*. Figure 5 illustrates taint tag spilling operation. If `R4` spills into memory, its taint tag will be stored in the next slot in the memory stack. Normally, the compiler will optimize register allocations to minimize register spilling. Therefore, runtime performance will not be affected too much. We will evaluate this performance overhead in Section 6.

Taint Tag of Object Fields The ART runtime maintains a heap for storing objects. The TAINTART runtime adds

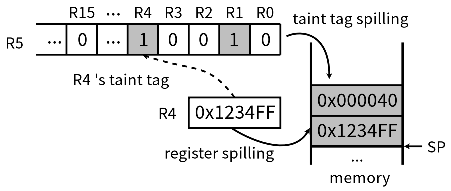


Figure 5: Taint tag spills into memory.

an extra space for each field in an object to indicate its taint status. For array, object array and string objects, we only store one taint tag to minimize the memory usage. We modified the heap allocator in the ART runtime to allocate spaces for storing taint tag. When there is an access to a field of an object, the TaintART compiler will load its taint tag from the memory to the corresponding register taint storage. We will presents taint propagation logic for field reading and writing in details later.

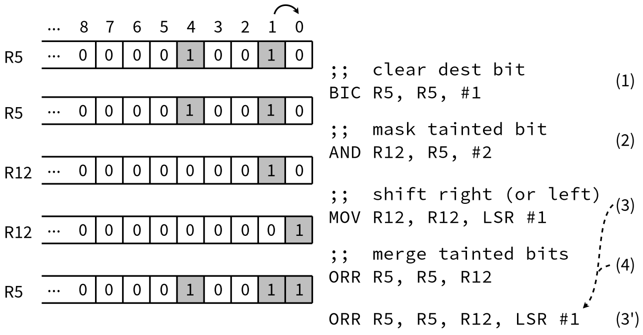
3.3 Taint Propagation Logic

TaintART provides variable-level taint tracking by instrumenting the Android compiler. Basically, TaintART tracks registers used for primitive type variables and object references. This section presents basic taint propagation logic, taint propagation via methods calls, and propagation between apps through binder IPC.

3.3.1 Taint Propagation Logic

The code builder of the ART compiler will transform the original dex bytecode into an internal control flow graph (`HGraph`). The dex instructions will be represented as `HInstruction` classes internally in the `HGraph`. For example, a `const/4` dex bytecode will be built as an `HIntConstant` instruction. The `HGraph` consists of many basic blocks (or `HBasicBlocks`). Then based on the `HGraph`, the optimizer will conduct various optimization strategies such as phi elimination, liveness analysis, dead code elimination, and constant folding. The ART compiler generator operates on `HGraph` and transforms internal instruction representation (`HInstruction`) in basic blocks into native code. The TaintART compiler instruments the original ART compiler and inserts code blocks to handle taint propagation logic.

The taint propagation logic is a set of operations (i.e., the `HInstruction` classes) which may cause variables' taint labels to change status. Table 1 shows descriptions of all types of tracking taint propagation logic including move operations, unary/binary operations, array operations, and field operations. The "HInstruction" column indicates classes of instructions. An `HInstruction` class contains its instruction type and related locations. The "Location" field is an abstraction over the potential registers containing variables or constants. For instance, the `HBinaryOperation` class contains a set of binary operations such as addition (`HAdd`), subtraction (`HSub`) and multiplication (`HMul`). There are three locations related to this type of instructions which are `first`, `second`, and `out`. The semantic of `HBinaryOperation` is to conduct an operation ($\otimes$) such as addition ($+$) on the `first` location and the `second` location, and store ($\leftarrow$) the final result in the `out` location. Note that because TaintART tracks taint propagation on optimized compiled code, compared to TaintDroid's VM-based taint logic, TaintART will introduce much less instructions on handling taint status changes. For instance, we do not need to track constant instructions (e.g., `const/4`, `const/16` and `const/high16` in dex bytecode) as taint logic propagation. This is because during the optimization of the ART compiler, the optimizer will conduct constant folding and inline the remaining constants into related instructions. Therefore, the code generator will ignore this type of instructions.



```
;;  clear dest bit
BIC R5, R5, #1                          (1)

;;  mask tainted bit
AND R12, R5, #2                         (2)

;;  shift right (or left)
MOV R12, R12, LSR #1                    (3)

;;  merge tainted bits
ORR R5, R5, R12                         (4)

ORR R5, R5, R12, LSR #1                 (3')
```

Figure 6: Taint tag propagates from R1 to R0 for the `MOV R0, R1` instruction.

Because TaintART uses CPU registers as the taint tag storage, the logic of taint label propagation is simple and fast. Figure 6 illustrates an example of the four steps to propagate taint status from an `R1` variable to an `R0` variable for the `MOV R0, R1` instruction. There are two registers involved. The `R5` register is the taint storage. We also need a temporary register for taint labels propagation. Fortunately, the ART compiler reserves the `R12` register and provides it for temporary usage. The four steps for taint propagation are (1) clear destination bit, (2) masking tainted bit, (3) shifting bits, and (4) merging tainted bits. Note that the last two steps can be combined into one instruction (3′) in ARM architecture devices. Therefore, TaintART only needs three data processing instructions without memory access to efficiently propagate a taint label. This is important because all instructions on taint propagation logic should be tracked at runtime. If it is not designed properly, this will introduce a huge impact on the runtime performance. We will perform instruction-level microbenchmak on Section 6. For binary operations (i.e., `HBinaryOperation`), we take the maximum value of taint tags from the `first` location and the `second` location and set it as the taint tag of the `out` location so as to achieve multi-level awareness. We will discuss the multi-level scenario in Section 4.

3.3.2 Method Invocation Taint Propagation

For the method invocation, we need to handle the taint propagation by passing values through method parameters. According to the method calling convention in ART environment, the `R1`, `R2` and `R3` registers are used for passing first three parameters. If the number of parameters is greater than three, the remaining parameters will be spilled into the memory. The TaintART compiler will push the taint storage register (i.e., `R5`) into the memory at the method frame entry. Then, all bits of `R5` are cleared except taint label bits for `R1`, `R2` and `R3` three passing parameters. For the spilled parameter registers, we do not need to do extra operations. Same as the taint tag spilling method we discussed in taint tag storage section, the system utilizes another word

Table 1: Descriptions of multi-level aware taint propagation logic.

HInstruction (Location)	Semantic	Taint Propagation Logic Description
HParallelMove(dest, src)	dest ← src	Set dest taint to src taint, if src is constant then clear dest taint
HUnaryOperation(out, in) HBooleanNot, HNeg, HNot	out ← in	Set out taint to in taint, unary operations ∈ {!, -, ~}
HBinaryOperation(out, first, second) HAdd, HSub, HMul, HDiv, HRem, HShl, HShr, HAnd, HOr, HXor	out ← first ⊗ second	Set out taint to max(first taint, second taint), ⊗ ∈ {+, -, *, /, %, <<, >>, &, |, ^}
HArrayGet(out, obj, index)	out ← obj[index]	Set out taint to obj taint
HArraySet(value, obj, index)	obj[index] ← value	Set obj taint to value taint
HStaticFieldGet(out, base, offset)	out ← base[offset]	Set out taint to base[offset] field taint
HStaticFieldSet(value, base, offset)	base[offset] ← value	Set base[offset] field taint to value taint
HInstanceFieldGet(out, base, offset)	out ← base[offset]	Set out taint to base[offset] field taint
HInstanceFieldSet(value, base, offset)	base[offset] ← value	Set base[offset] field taint to value taint

to store taint tag and loads into taint tag storage register when the spilled registers is needed. At the frame exit, R5 will be popped and restored finally.

3.3.3 *Binder IPC & Native Code Taint Propagation*

Binder mechanism is one common way to exchange messages between apps. Because the binder implementation in Android framework is stable, we employ previous methodology. We track taint tag propagation in message-level granularity for performance concern. We add an extra field in binder parcel to indicate the taint status of this message. When sending a binder message, the TaintART runtime will append taint status into the parcel message and unpack the taint status when a message arrives. Because TaintART mainly focus on tracking information flows within ART environment. We can employ existing work such as NDroid [42] which mainly focuses on tracing information flow through JNI.

4. IMPLEMENTATION

In this section, we discuss implementations on taint sources and sinks, taint interface library and some deployment details of TaintART.

4.1 Taint Sources and Sinks

Taint analysis system is a general methodology which is widely used in vulnerability detection, privacy tracking, and malware analysis. Based on our design, we implement TaintART for tracking multi-level information flows. This will help users to monitor sensitive information and assist analysts to dissect malware behaviors. Moreover, we can also adopt TaintART for enforcing policy on sensitive data leakage.

In our implementation, we track four types of data from fifteen different sources. Table 2 lists all data sources. We categorize them as device identity, sensor data, sensitive content and location data. We place taint source logic in corresponding classes to track these data. For example, device identity such as IMEI number can be obtained from TelephonyManager. TelephonyManager is one of many system services in system server process. Apps can acquire telephony data by sending request message to the telephony manager through binder IPC. Therefore, our sources are placed at a method in the TelephonyManager class for handling requests of device identity from binder. The TaintART source logic will attach a taint tag in that binder parcel.

For sink placements, we consider leakage to network and external storage. If tainted data is passed to sink functions, TaintART will record this event as data leakage. Because we provide interfaces for placing sources and sinks, analysts can focus and track data they are interested in.

In Section 3, we use one bit for taint tag to explain our system design. For some scenarios, we need to track multiple data sources. Therefore, we can use more bits for taint storage to represent multiple taint tag states. For tracking sensitive data in multiple levels, we use two bits for storing one taint tag. We categorize data leakage in four levels. The level zero indicates that there is data leakage. Because device identity information are always used for advertisement tracking and account identity in current apps, these data are less sensitive and are classified in the first level. Sensor data such as accelerometer and rotation information is in the second level. At last, location data and sensitive content such as messages, contact lists and call logs are categorized in the third level. We consider data in level three as the most sensitive data.

4.2 Taint Analysis Interface

Because TaintART is designed for general data flow analysis, analysts can develop new tools or services based on TaintART. We provide two basic interfaces for taint analysis which are addTaint() and getTaint(). Developers can use addTaint() to update taint tag of a specific local variable or objects, and inspect taint tag later. To achieve better performance, we implement these two primitive interfaces as intrinsic functions so that the TaintART compiler can inline the functions at the compilation time.

4.3 Implementation & Deployment Details

We prototype TaintART based on the current latest Android version (Android 6.0.1 Marshmallow AOSP tag android-6.0.1_r1) for Nexus 5 (target aosp_hammerhead). We customize the ART compiler and ART runtime sources to implement taint tag propagation. We also add sources tracking logic in Android framework sources. To support taint propagation through JNI, we also customize binder-related sources in Android framework. In summary, we provide customized binary and libraries such as dex2oat,

Table 2: Taint Sources and Privacy Leakage Levels

Level	Leaked Data	Source	Class/Service
0 (00)	No Leakage	N/A	N/A
1 (01)	Device Identity	IMSI	`TelephonyManager`
		IMEI	`TelephonyManager`
		ICCID	`TelephonyManager`
		SN	`TelephonyManager`
2 (10)	Sensor Data	Accelerometer	`SensorManager`
		Rotation	`SensorManager`
	Location Data	GPS Location	`LocationManager`
		Last Seen Location	`LocationManager`
		Network Location	`LocationManager`
3 (11)	Sensitive Content	SMS	`ContentResolver`
		MMS	`ContentResolver`
		Contacts	`ContentResolver`
		Call log	`ContentResolver`
		File content	`File`
		Camera	`Camera`
		Microphone	`MediaRecorder`

`libart.so` and `libart-compiler.so`. For the implementation, we reuse the peer-reviewed code from AOSP as much as possible to ensure the stability and security of TaintART. Since the code base of ART environment is stable after Android 5.0, our implementation is generic for Android 5.0 and 6.0 versions.

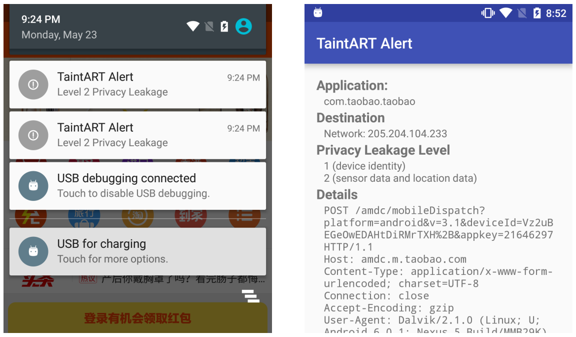
To deploy TaintART on devices, we do not require users or analysts to reinstall the customized systems from scratch. Instead, analysts can overwrite our customized binary and libraries to a target device with root privilege. Because we modified Android framework code to place taint sources, TaintART needs to re-compile framework code again so that taint tags can transit through the framework code.

5. CASE STUDY

In this section, we study several popular apps and analyze the possibility of privacy leakage using TaintART.

Experimental Setup We download and compile the latest TaintDroid targeting `aosp_arm-eng`, which is based on Android 4.3 released in July 2013 (`android-4.3_r1`). We run our TaintART system on Android 6.0.1 which was released in December 2015 (`android-6.0.1_r1`). Apps in our dataset used for the case study are downloaded from the Google official market (Google Play) in May 2016.

Privacy Tracking We test popular apps to study potential privacy leakage in the "top chart" for various categories including shopping, payment, social, entertainment, etc. We execute and manually interact with each app in TaintDroid and TaintART respectively and record the reports of privacy leakage. Table 3 illustrates details of our analysis. By default, TaintDroid will deny loading all external native libraries. This makes some apps crashed at launch time. We comment out those codes and allow JNI invocations for TaintDroid. However, some functions of Taobao and Alipay are broken because of compatibility issues. One reason is that they try to use



9:24 PM
Monday, May 23

TaintART Alert 9:24 PM
Level 2 Privacy Leakage

TaintART Alert 9:24 PM
Level 2 Privacy Leakage

USB debugging connected
Touch to disable USB debugging.

USB for charging
Touch for more options.

TaintART Alert

Application:
com.taobao.taobao

Destination
Network: 205.204.104.233

Privacy Leakage Level
1 (device identity)
2 (sensor data and location data)

Details

```
POST /amdc/mobileDispatch?
platform=android&v=3.1&deviceId=Vz2uB
EGeOwEDAHtDiRMrTXH%2B&appkey=21646297
HTTP/1.1
Host: amdc.m.taobao.com
Content-Type: application/x-www-form-
urlencoded; charset=UTF-8
Connection: close
Accept-Encoding: gzip
User-Agent: Dalvik/2.1.0 (Linux; U;
Android 6.0.1; Nexus 5 Build/MMB29K)
```

Figure 7: Screenshots of privacy policy enforcement.

`MultiDex` class in their apps and this interface is not supported in Android 4.3. For privacy tracking, we found that shopping apps such as Taobao and JD.COM accesses device identity and sensor data. By inspecting the outbound packets, we found some packets sent to remote server contain tainted identity and sensor data. For example, all HTTP requests of JD.COM contain a device id field: "`client.action?functionId=jshopUrlAdapter&body=%7B%7D&uuid=[IMEI]&clientVersion=5.1.0`". Taobao will include device orientation information in the "User-Agent" field for all requests. Both TaintDroid and TaintART can capture these leakage events. It is worth noting that the latest Facebook app no longer supports Android version less than 5.0. Therefore, TaintDroid cannot analyze new Facebook app. This shows TaintART is more versatile.

Policy Enforcement Because TaintART supports the latest Android runtime and provides an efficient, extensible, as well as easy-to-deploy methodology, it is also suitable for policy enforcement. Unlike systems enforcing sensitive API invocations, TaintART knows the sensitivity of data passing to enforced functions. Based on TaintART, we prototype a privacy policy enforcement function. In essence, users can pre-define multi-level policy rules. For instance, we have already defined four privacy leakage levels in the previous section. For each level, users can define different policies. Table 4 shows policies in four levels. We only record level 1 data leakage in a log because many apps use on device identity for authentication. For level 2, we alert users with a notification and replace sensitive information as random values. Figure 7 depicts a screenshot of a notification of the level 2 privacy leakage event. For sensitive data in level 3, TaintART will block any access to the data. Finally, we also provide interfaces for developers to customize actions for different policies.

6. EVALUATION

In this section, we perform macrobenchmarks for common apps and microbenchmarks for the compiler, Java environment, and investigate the memory usage of TaintART. We also evaluate the compatibility of TaintART against Android official compatibility test suite. The device used in our evaluation is a Nexus 5 device with Quadcore 2.3 GHz CPU, 2 GB memory and 16 GB internal storage. The test device runs the Google official Android firmware, which is Marshmallow 6.0.1 with build number `MMB29K` and the Linux kernel version 3.4.0.

Table 3: Privacy Leakage Analysis on Popular Apps.

App Name	Version	Min/Target SDK	TaintDroid Result (Error Message)	TaintART Result
Taobao	5.7.2	14/23	Some functions are broken: "cannot find method" in config error	2: device identity, sensor data, location data
Alipay	9.6.6.051201	15/23	Cannot login: "It is crowed" error	2: device identity, sensor data, location data
JD.COM	5.1.0	14/14	Device identity and accelerometer leakage	2: device identity, sensor data, location data
Facebook	77.0.0.20.66	21/23	Cannot install: the minimum SDK is Android 5.0	1: device identity
Skype	6.34.0.715	15/23	Device identity leakage	1: device identity
Instagram	8.1.0	16/23	Device identity leakage	1: device identity
Spotify	5.3.0.995	15/23	No leakage	0: no leakage
Amazon Shopping	6.6.0.100	11/23	No leakage	0: no leakage

Table 4: Privacy Enforcement Policy.

Level	Description of Enforcement Policy
0	N/A
1	record events
2	record events, alert users and rewrite sensitive information
3	record events, alert users and prevent accesses

Table 5: Macrobenchmark Results.

Macrobenchmark Name (ms)	Original (with Optimizing Backend)	TaintART
App Launch Time	348.2	370.3
App Installation Time	1680.5	1886.3
Contacts Read/Write	7.0/9538.5	8.4/9655.2

6.1 Macrobenchmarks

Because TaintART is a general framework that can be used by end-users to protect their privacy, we perform several macrobenchmarks to measure the overhead for normal usage. The evaluation results are shown in Table 5.

We first evaluate the app's load time. We create an app based on Android 6.0.1 SDK with one activity (generated by the app template of Android Studio 2.0 with Gradle 1.2.3). We use an Android UI/application exerciser (i.e., the `Monkey` tool [26]) to launch this app and record the time (t_0). When the `attachBaseContext` method is called, which means the activity has been displayed on the screen, we record the time (t_1). Therefore, $t_1 - t_0$ represents the elapsed time from the launch time into app context. The result indicates 22.1 ms (6.0 %) overhead on app's launch time. The overhead is clearly acceptable because most of the logic for launching an app is executed in the native code, and TaintART mainly affects runtime performance on the Java environment. For the installation time, TaintART introduces about 205.8 ms (12.2 %) overheads, which are mainly attributed to the instrumented ART compiler. We will present the evaluation result of the compiler microbenchmark in the next subsection. Since we add taint sources on the content resolver, we also evaluate the performance of reading and write contacts in address book so as to demonstrate the impact. We first write 100 contacts with full information in batch through the content resolver, then query these inserted contact names for 100 times. The result shows that there are 20 % and 12 % overhead on read and write respectively. In summary, TaintART introduces an acceptable level of overhead to end-users or to analysts if they want to understand the information flows.

6.2 Microbenchmarks

To understand the performance for some major components in TaintART, we perform microbenchmarks on the compiler and Java runtime. We also investigate the memory usage and inter-process communication cost of TaintART.

Compiler Microbenchmarks Because TaintART instruments ART compiler and inserts taint logic at compilation time, we evaluate the number of instructions and the overall compilation time. For compiler microbenchmark, we utilizes all 80 built-in apps in AOSP which can be found in `/out/target/common/obj/APPS/` as our evaluation dataset including calculator, contacts, browser, download manager, etc. We compile all apps using the original compiler and TaintART compiler respectively and record the time of compilation. Figure 8 illustrates the compilation time for 80 built-in apps. By adopting the TaintART compiler, the average time increases from 336.076 milliseconds to 403.064 milliseconds and introduces about 19.9 % overhead. Because Android uses ahead-of-time compilation strategy, an app is only compiled once at the installation time. Therefore, the overhead on compilation time is acceptable for analysis usage. In addition, we use `oatdump` to disassemble compiled native code and categorize instructions into seven types. Figure 9 depicts the total number of instructions for all 80 apps and the numbers in different categories. The total number of instructions increases about 21 %. The increases are mainly in data processing instructions (Type II) including arithmetic instructions (`ADD`, `SUB`), logical instructions (`ORR`, `AND`), movement instructions (`MOV`, `MVN`), etc. For memory access instructions (Type I) which will cost more CPU cycles, TaintART compiler only introduces about 0.8 % more instructions. Because of this, the overhead of runtime performance of TaintART is minimal, as we will show in the Java microbenchmark later. This means that TaintART can achieve better runtime performance than the VM-based TaintDroid with the gains of AOT compilation strategy in the new ART environment.

Java Microbenchmark Because Android apps are mainly written in Java and TaintART tracks information flows in

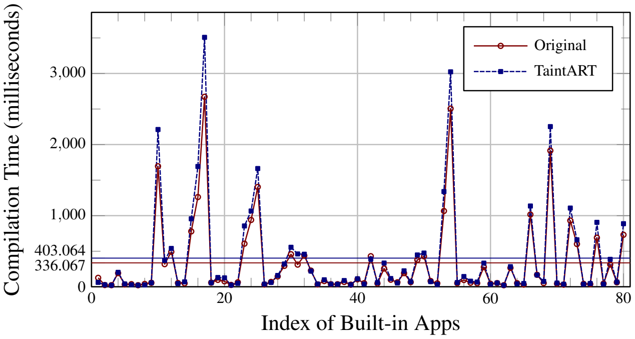


Figure 8: Comparison of compilation time.

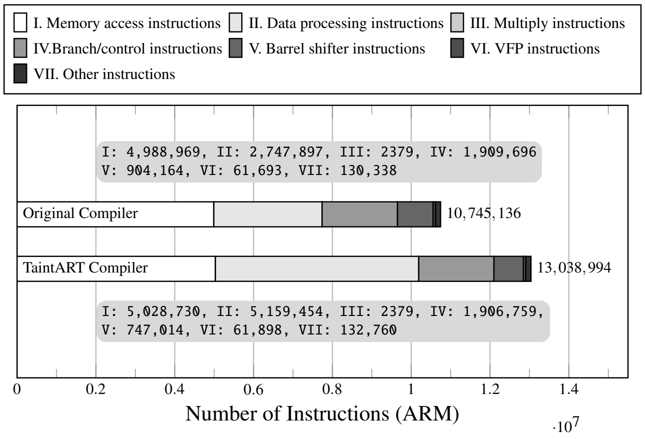


Figure 9: Comparison of instruction numbers for different types.

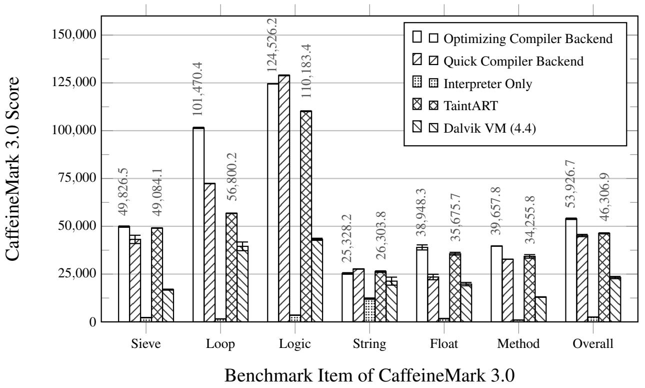


Figure 10: CaffeineMark 3.0 Java microbenchmark. Error bars indicate 95 % confidence intervals.

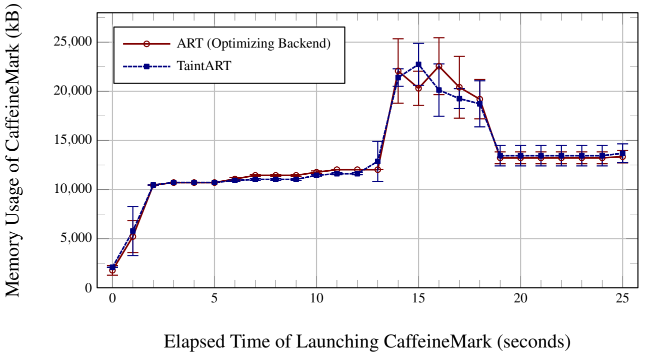


Figure 11: CaffeineMark 3.0 memory microbenchmark. Error bars indicate 95 % confidence intervals.

apps, Java microbenchmark can accurately reflect the runtime overhead introduced by TAINTART. We utilize CaffeineMark 3.0 Java benchmark tools [39] to evaluate six types of operations including sieve, loop, logic, string, float and method call. Note that the scores of CaffeineMark 3.0 are only useful for relative comparison. For comparison, we evaluate and record scores in five different runtime environments on the same Nexus 5 device: (1) ART with optimizing compiler backend, (2) ART with quick compiler backend, (3) interpreter only, (4) TAINTART compiler, and (5) Dalvik in Android 4.4. We run the tool ten times and record scores for each sub-benchmark. Figure 10 illustrates scores for each environment. Compared to the original ART compiler with optimizing backend, TAINTART compiler introduces about 14 % overhead overall and it is comparable with its predecessor TaintDroid. Most importantly, we notice that ART brings a huge improvement over the legacy Dalvik environment. Compared to the legacy Dalvik environment without any instrumentation, TAINTART can achieve about 99.8 % more scores for overall runtime performance.

Memory Microbenchmark We also perform the memory microbenchmark. We run the CaffeineMark 3 Java benchmark tool ten times and monitor the `/proc/[pid]/status` file at runtime. Figure 11 shows the virtual memory resident set (VmRSS) size at runtime representing the portion of memory occupied by the benchmark process in memory. Because TAINTART mainly relies on CPU registers for taint tag storage, the overhead on memory usage is only about 0.4 %. For comparison, TaintDroid introduces 4.4 % memory overhead mainly because it doubles the size of internal stack of Dalvik VM for storing taint tags, and we avoid this by carefully using the register resources.

IPC Microbenchmark To carry out the IPC benchmark of TAINTART, we developed client and server applications which communicate through binder. The client app will send messages to server for setting and getting an object. The object contains a string and an integer field. We continuously conduct pair of getting and setting requests for ten thousand times and record their execution times. We also average memory usages for client app and server app during the communication phase. Table 6 depicts the benchmark results. There are about 4.35 % overhead on IPC execution, and less than 4 % memory overhead.

6.3 Compatibility Evaluation

We also evaluate the compatibility of TAINTART using Android Compatibility Test Suite (CTS) in `6.0_r5` version. We execute the standard CTS plan which contains 131 test package in total. We select several system, runtime and security test packages and illustrate some partial results (due to page limit) in Table 7. Both TAINTART and original devices failed on the same 186 cases among 100317 cases. These failed cases are relevant and mainly caused by environmental setups such SD card or SIM card. There are three

Table 6: IPC Throughput Benchmark (10,000 pairs of messages).

Macrobenchmark Name	Original	TaintART	Overhead
Execution Time	2987 ms	3117 ms	4.35 %
Memory (client)	51 572 kB	53 170 kB	3.10 %
Memory (server)	38 812 kB	39 689 kB	2.26 %

Table 7: Android Compatibility Test Suite (CTS) results. The prefixes on test package names were removed for simple representations.

Test Package	# of Tests	Tests Failed	
		Android	TaintART
`app`	266	6	6
`content`	619	0	0
`bionic`	1274	0	0
`libcore`	23 371	0	0
`database`	264	0	0
`location`	99	0	0
`os`	409	0	0
`telephony`	67	6	6
`util`	206	0	0
`security`	103	0	3
others	100 317	156	159
Total (131 packages)	126 995	186	189

more failed cases introduced by TaintART in security test package. The reason is that TaintART needs root privilege to deploy on stock devices. In summary, TaintART customizes the Android runtime and compiler, but maintains similar compatibility level when compare with the original Android environment.

7. DISCUSSION

Limitations Here, we discuss some limitations of TaintART (and we like to state that other dynamic taint analysis systems have the same limitations). Firstly, TaintART cannot effectively track implicit data flows. This means that attackers can find ways [29, 45] to exchange data without the track of TaintART. Because of the limitation of taint analysis methodology, we cannot exhaustively monitor all implicit leakage. Our goal is to track common explicit data flows for privacy leakage analysis and increase the bar for malware writers. Secondly, malware can utilize some anti-analysis techniques [40] to detect host devices. For example, malware can infer the running system by inspecting the size of compilation binary file. Thirdly, for malware analysis, analysts need to manually trigger the behaviors. However, researchers [54] have proposed various methodologies to generate input for dynamic analysis.

TaintART on Other Architectures Because most mobile devices are based on the ARM architecture (99 % according to [7]), our TaintART prototype is implemented on an 32-bit ARM-based device. For the ARM64 (AArch64) architecture, it provides 31 general-purpose registers. We can utilize three of them (i.e., X25, X26, and X27) for taint tags (tracking 24 registers for data storage in Android). In this case, we can obtain eight bits for each tag to store more semantics. In addition, AArch64 also has a set of instructions (e.g., UBFX) for moving and copying bits among registers. This will make propagation logic much easier and faster. Moreover, due to availability of registers, performance overhead will be comparable with (even better than) 32-bit architecture. In addition, the latest Android version supports other architectures including x86-64. To support other architectures, we plan to port the ARM code generator of TaintART compiler to other code generators so as to utilize architecture-specific features. This is our future work.

8. RELATED WORK

With the rapid growth of mobile users, hackers and researchers investigated many severe vulnerabilities and proposed mitigation solutions [20, 28, 47, 66, 64, 18, 56, 13, 65, 44, 32, 48, 60, 6, 49, 4] in current mobile ecosystem. To understand the hidden malicious behaviors of malware such as stealing private and sensitive information, researchers proposed app analysis methodologies via dynamic and static perspective. Based on these methodologies, some runtime policy enforcement systems are proposed to prevent malicious events or privacy leakage.

Dynamic Analysis System There are many systems which dynamically monitor runtime information in different layers of the system. DroidScope [59], BareCloud [34] and CopperDroid [51] introspect Dalvik VM to capture dynamic information for reconstructing malware behaviors. VetDroid [61] analyzes permission usages to find information leaks and identifies subtle vulnerabilities of apps. Poeplau et al. [41] systematically analyze malicious dynamic code loading by a customized Dalvik VM. Similar to virtual machine introspection technique, to reduce privacy leakage across apps (unregulated aggregation), LinkDroid [21] analyzes app links across different apps dynamically. Note that these systems are proposed for monitoring malicious behaviors, they cannot track information-flow which can accurately detect privacy leakage. Minemu [8] is a general dynamic taint analysis system based on an optimized emulator with JIT compilation. Similar with Minemu, TaintDroid [19] is the most relevant information-flow tracking system for Android. Based on TaintDroid, NDroid [42] can further track information flows through JNI by customizing Android emulators (QEMU). As we discussed in previous sections, TaintDroid is based on legacy runtime and cannot be ported to current environments, which makes it impossible to analyze apps developed for the latest Android systems.

Static Analysis System Many systems utilize disassembled code and try to precisely model runtime behavior and use program analysis technique to resolve information flows. Lu et al. [37] proposes CHEX framework to detect component hijacking by computing data flows using Wala[22]. AndroidLeaks [23] detects potential privacy leaks on a large scale. FlowDroid [2] can preform more precise context, flow, field, object-sensitive and lifecycle-aware analysis. ComDroid [14], AmanDroid [53], R-Droid [3], IccTA [35] and HornDroid [10] try to improve the static analyzer to detect implicit data flows across components among Android apps. Based on call graph, EdgeMiner [11] can automatically generate API summaries to detect implict control flow transitions through the Android framework. DroidSafe [27] models runtime using an accurate analysis stubs technique so as to capture missing semantic events such as life-cycle events and callback context in the static code. AAPL [36] can detect privacy leaks by combining multiple special static analysis techniques and purify results by employing a novel peer voting technique. AppAudit [57] combines static and dynamic analysis to reduce the over-estimating problem introduced by static taint analysis. RiskMon [33] can assess apps' risk by adopting machine leaning algorithm. Note that these systems can analyze large number of apps in an offline manner, but without executing apps, the static analysis technique cannot track the realtime data flows and privacy leakage.

Policy Enforcement System To detect suspicious behaviors and prevent potential privacy leakage, researchers proposed many policy enforcement systems for Android. Aurasium [58] and RetroSkeleton [17] can add enforcement policies and fine-grained mandatory access control on sensitive API invocations by rewriting and repackaging apps. However, hackers may bypass these policies due to the incomplete app rewriting [30]. Besides using repackage technique, systems like FlaskDroid [9], Patronus [50], ARTDroid [15], and ASM [31] can achieve fine-grained mandatory access control by hooking Android system services and low level system calls. With the similar technique, DeepDroid [52] mainly focus on policy enforcement under enterprise domain. Airbag [55] can provides an sandbox environment which is resistant to malware infection for legitimate apps. Afonso et al. [1] create a sandboxing policy for Android native code. These systems add policy for each sensitive API calls, but still cannot differentiate legitimate or malicious behavior. Because TaintART can track information flows, our system can accurately detect data leakage and alert users at runtime.

9. CONCLUSION

In this paper, we design a compiler-instrumented information-flow analysis platform called TaintART on the new Android ART environment. We adopt dynamic taint analysis methodology for tracking sensitive data. TaintART instruments the ART compiler and runtime for handling taint propagation logic, tracking source methods and report data leakage from sink methods. TaintART employs CPU registers for multi-level taint tag to minimize storage and achieve fast taint propagation logic. We also provide APIs for analysts to track specific data. Based on this platform, we implement a multi-level privacy tracking system which can be used for policy enforcement. Our evaluation results show that TaintART introduces less than 15 % overheads on an overall CPU-bound microbenchmark and imposes negligible overhead on built-in and third-party apps. Additionally, compared to legacy Dalvik environment in Android 4.4, the TaintART system can achieve two times faster performance for Java runtime benchmark.

10. REFERENCES

[1] V. Afonso, A. Bianchi, Y. Fratantonio, A. Doupé, M. Polino, P. de Geus, C. Kruegel, and G. Vigna. Going native: Using a large-scale analysis of android apps to create a practical native-code sandboxing policy. In *NDSS*, 2016.

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

[3] M. Backes, S. Bugiel, E. Derr, S. Gerling, and C. Hammer. R-droid: Leveraging android app analysis with static slice optimization. In *ASIACCS*, 2016.

[4] M. Backes, S. Bugiel, E. Derr, P. McDaniel, D. Octeau, and S. Weisgerber. On demystifying the android application framework: Re-visiting android permission specification analysis. In *USENIX Security*, 2016.

[5] R. Balebako, J. Jung, W. Lu, L. F. Cranor, and C. Nguyen. Little brothers watching you: Raising awareness of data leaks on smartphones. In *SOUPS*, 2013.

[6] A. Bianchi, J. Corbetta, L. Invernizzi, Y. Fratantonio, C. Kruegel, and G. Vigna. What the app is that? deception and countermeasures in the android user interface. In *S&P*, 2015.

[7] Bloomberg. Arm designs one of the world's most-used products. http://www.bloomberg.com/bw/articles/2014-02-04/arm-chips-are-the-most-used-consumer-product-dot-where-s-the-money.

[8] E. Bosman, A. Slowinska, and H. Bos. Minemu: The world's fastest taint tracker. In *RAID*, 2011.

[9] S. Bugiel, S. Heuser, and A.-R. Sadeghi. Flexible and fine-grained mandatory access control on android for diverse security and privacy policies. In *USENIX Security*, 2013.

[10] S. Calzavara, I. Grishchenko, and M. Maffei. Horndroid: Practical and sound static analysis of android applications by smt solving. In *Euro S&P*, 2016.

[11] Y. Cao, Y. Fratantonio, A. Bianchi, M. Egele, C. Kruegel, G. Vigna, and Y. Chen. Edgeminer: Automatically detecting implicit control flow transitions through the android framework. In *NDSS*, 2015.

[12] J. Chen, H. Chen, E. Bauman, Z. Lin, B. Zang, and H. Guan. You shouldn't collect my secrets: Thwarting sensitive keystroke leakage in mobile ime apps. In *USENIX Security*, 2015.

[13] Q. A. Chen, Z. Qian, and Z. M. Mao. Peeking into your app without actually seeing it: Ui state inference and novel android attacks. In *USENIX Security*, 2014.

[14] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing inter-application communication in android. In *MobiSys*, 2011.

[15] V. Costamagna and C. Zheng. Artdroid: Simple and easy to use library to intercept virtual-method calls under the android art runtime. In *Proceedings of the Workshop on Innovations in Mobile Privacy and Security*, 2016.

[16] M. Dam, G. Le Guernic, and A. Lundblad. Treedroid: A tree automaton based approach to enforcing data processing policies. In *CCS*, 2012.

[17] B. Davis and H. Chen. Retroskeleton: retrofitting android apps. In *MobiSys*, 2013.

[18] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel. An empirical study of cryptographic misuse in android applications. In *CCS*, 2013.

[19] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *TOCS*, 2014.

[20] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin. Permission re-delegation: Attacks and defenses. In *USENIX Security*, 2011.

[21] H. Feng, K. Fawaz, and K. G. Shin. Linkdroid: reducing unregulated aggregation of app usage behaviors. In *USENIX Security*, 2015.

[22] S. Fink and J. Dolby. Wala–the tj watson libraries for analysis, 2012.

[23] C. Gibler, J. Crussell, J. Erickson, and H. Chen. Androidleaks: automatically detecting potential privacy

leaks in android applications on a large scale. In *TRUST*, 2012.

[24] Google. Android dashboards. https://developer.android.com/about/dashboards/index.html.

[25] Google. Dalvik jit. http://android-developers.blogspot.hk/2010/05/dalvik-jit.html.

[26] Google. Ui/application exerciser monkey. https://developer.android.com/studio/test/monkey.html.

[27] M. I. Gordon, D. Kim, J. H. Perkins, L. Gilham, N. Nguyen, and M. C. Rinard. Information flow analysis of android applications in droidsafe. In *NDSS*, 2015.

[28] M. C. Grace, Y. Zhou, Z. Wang, and X. Jiang. Systematic detection of capability leaks in stock android smartphones. In *NDSS*, 2012.

[29] gsbabil. Antitaintdroid.

[30] H. Hao, V. Singh, and W. Du. On the effectiveness of api-level access control using bytecode rewriting in android. In *ASIACCS*, 2013.

[31] S. Heuser, A. Nadkarni, W. Enck, and A.-R. Sadeghi. Asm: A programmable interface for extending android security. In *USENIX Security*, 2014.

[32] H. Huang, S. Zhu, K. Chen, and P. Liu. From system services freezing to system server shutdown in android: All you need is a loop in an app. In *CCS*, 2015.

[33] Y. Jing, G.-J. Ahn, Z. Zhao, and H. Hu. Towards automated risk assessment and mitigation of mobile applications. *TDSC*, 2015.

[34] D. Kirat, G. Vigna, and C. Kruegel. Barecloud: bare-metal analysis-based evasive malware detection. In *USENIX Security*, 2014.

[35] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel. Iccta: Detecting inter-component privacy leaks in android apps. In *ICSE*, 2015.

[36] K. Lu, Z. Li, V. P. Kemerlis, Z. Wu, L. Lu, C. Zheng, Z. Qian, W. Lee, and G. Jiang. Checking more and alerting less: Detecting privacy leakages via enhanced data-flow analysis and peer voting. In *NDSS*, 2015.

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

[38] W. Meng, R. Ding, S. P. Chung, S. Han, and W. Lee. The price of free: Privacy leakage in personalized mobile in-app ads. In *NDSS*, 2016.

[39] Pendragon Software Corporation. CaffeineMark 3.0. http://www.benchmarkhq.ru/cm30/.

[40] T. Petsas, G. Voyatzis, E. Athanasopoulos, M. Polychronakis, and S. Ioannidis. Rage against the virtual machine: hindering dynamic analysis of android malware. In *EuroSec*, 2014.

[41] S. Poeplau, Y. Fratantonio, A. Bianchi, C. Kruegel, and G. Vigna. Execute this! analyzing unsafe and malicious dynamic code loading in android applications. In *NDSS*, 2014.

[42] C. Qian, X. Luo, Y. Shao, and A. T. Chan. On tracking information flows through jni in android applications. In *DSN*, 2014.

[43] V. Rastogi, Y. Chen, and W. Enck. Appsplayground: automatic security analysis of smartphone applications. In *CODASPY*, 2013.

[44] C. Ren, Y. Zhang, H. Xue, T. Wei, and P. Liu. Towards discovering and understanding task hijacking in android. In *USENIX Security*, 2015.

[45] G. Sarwar, 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 *SECRYPT*, 2013.

[46] 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 *S&P*, 2010.

[47] Y. Shao, J. Ott, Q. A. Chen, Z. Qian, and Z. M. Mao. Kratos: Discovering inconsistent security policy enforcement in the android framework. In *NDSS*, 2016.

[48] M. Sun, M. Li, and J. C. S. Lui. Droideagle: Seamless detection of visually similar android apps. In *WiSec*, 2015.

[49] M. Sun, J. C. S. Lui, and Y. Zhou. Blender: Self-randomizing address space layout for android apps. In *RAID*, 2016.

[50] M. Sun, M. Zheng, J. C. S. Lui, and X. Jiang. Design and implementation of an android host-based intrusion prevention system. In *ACSAC*, 2014.

[51] K. Tam, S. J. Khan, A. Fattori, and L. Cavallaro. Copperdroid: Automatic reconstruction of android malware behaviors. In *NDSS*, 2015.

[52] X. Wang, K. Sun, Y. Wang, and J. Jing. Deepdroid: Dynamically enforcing enterprise policy on android devices. In *NDSS*, 2015.

[53] F. Wei, S. Roy, X. Ou, et al. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. In *CCS*, 2014.

[54] M. Y. Wong and D. Lie. Intellidroid: A targeted input generator for the dynamic analysis of android malware. In *NDSS*, 2016.

[55] C. Wu, Y. Zhou, K. Patel, Z. Liang, and X. Jiang. Airbag: Boosting smartphone resistance to malware infection. In *NDSS*, 2014.

[56] L. Wu, M. Grace, Y. Zhou, C. Wu, and X. Jiang. The impact of vendor customizations on android security. In *CCS*, 2013.

[57] M. Xia, L. Gong, Y. Lyu, Z. Qi, and X. Liu. Effective real-time android application auditing. In *S&P*, 2015.

[58] R. Xu, H. Saïdi, and R. Anderson. Aurasium: Practical policy enforcement for android applications. In *USENIX Security*, 2012.

[59] L. K. Yan and H. Yin. Droidscope: seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. In *USENIX Security*, 2012.

[60] X. Zhang, K. Ying, Y. Aafer, Z. Qiu, and W. Du. Life after app uninstallation: Are the data still alive? data residue attacks on android. In *NDSS*, 2016.

[61] Y. Zhang, M. Yang, B. Xu, Z. Yang, G. Gu, P. Ning, X. S. Wang, and B. Zang. Vetting undesirable behaviors in android apps with permission use analysis. In *CCS*, 2013.

[62] Y. Zhang, M. Yang, B. Zhou, Z. Yang, W. Zhang, and B. Zang. Swift: A register-based jit compiler for embedded jvms. In *VEE*, 2012.

[63] C. Zheng, S. Zhu, S. Dai, G. Gu, X. Gong, X. Han, and W. Zou. Smartdroid: an automatic system for revealing ui-based trigger conditions in android applications. In *SPSM*, 2012.

[64] M. Zheng, M. Sun, and J. C. S. Lui. Droidanalytics: a signature based analytic system to collect, extract, analyze and associate android malware. In *TrustCom*, 2013.

[65] M. Zheng, M. Sun, and J. C. S. Lui. Droidray: a security evaluation system for customized android firmwares. In *ASIACCS*, 2014.

[66] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *S&P*, 2012.