TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones

WILLIAM ENCK, North Carolina State University
PETER GILBERT, Duke University
SEUNGYEOP HAN, University of Washington
VASANT TENDULKAR, North Carolina State University
BYUNG-GON CHUN, Seoul National University
LANDON P. COX, Duke University
JAEYEON JUNG, Microsoft Research
PATRICK MCDANIEL, The Pennsylvania State University
ANMOL N. SHETH, Technicolor Research

Today's smartphone operating systems frequently fail to provide users with visibility into how third-party applications collect and share their private data. We address these shortcomings with TaintDroid, an efficient, system-wide dynamic taint tracking and analysis system capable of simultaneously tracking multiple sources of sensitive data. TaintDroid enables realtime analysis by leveraging Android's virtualized execution environment. TaintDroid incurs only 32% performance overhead on a CPU-bound microbenchmark and imposes negligible overhead on interactive third-party applications. Using TaintDroid to monitor the behavior of 30 popular third-party Android applications, in our 2010 study we found 20 applications potentially misused users' private information; so did a similar fraction of the tested applications in our 2012 study. Monitoring the flow of privacy-sensitive data with TaintDroid provides valuable input for smartphone users and security service firms seeking to identify misbehaving applications.

Categories and Subject Descriptors: D.4.6 [**Operating Systems**]: Security and Protection

General Terms: Design, Security, Performance

Additional Key Words and Phrases: Information-flow tracking, privacy monitoring, smartphones, mobile apps

ACM Reference Format:
Enck, W., Gilbert, P., Han, S., Tendulkar, V., Chun, B.-G., Cox, L. P., Jung, J., McDaniel, P., and Sheth, A. N. 2014. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Trans. Comput. Syst.* 32, 2, Article 5 (June 2014), 29 pages.
DOI: http://dx.doi.org/10.1145/2619091

This manuscript is an extension of the conference version appearing in the *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI'10)* [Enck et al. 2010]. This manuscript presents a more detailed description of the system design, implementation, and evaluation.
This work was supported in part by the Engineering Research Center of Excellence Program of Korea Ministry of Science, ICT & Future Planning (MSIP)/National Research Foundation of Korea (NRF) grant NRF-2008-0062609.
Author's addresses: W. Enck, Computer Science Department, North Carolina State University, Raleigh, NC; P. Gilbert, Computer Science Department, Duke University, Durham, NC; S. Han, Computer Science and Engineering Department, University of Washington, Seattle, WA; V. Tendulkar, Computer Science Department, North Carolina State University, Raleigh, NC; B.-G. Chun (corresponding author), Computer Science and Engineering Department, Seoul National University, Seoul, South Korea; email: bgchun@snu.ac.kr; L. P. Cox, Computer Science Department, Duke University, Durham, NC; J. Jung, Microsoft Research; P. McDaniel, The Pennsylvania State University, State College, PA; A. N. Sheth, Technicolor Research.

DOI: http://dx.doi.org/10.1145/2619091

1. INTRODUCTION

A key feature of modern smartphone platforms is a centralized service for downloading third-party applications. The convenience to users and developers of such "app stores" has made mobile devices more fun and useful, and has led to an explosion of development. Apple's App Store alone served 40 billion application downloads since its launch [Apple, Inc. 2013]. Many of these applications combine data from remote cloud services with information from local sensors such as a GPS receiver, camera, microphone, and accelerometer. Applications often have legitimate reasons for accessing this privacy-sensitive data, but users would also like assurances that their data is used properly. Incidents of developers relaying private information back to the cloud [Moren 2009; Davies 2009] and the privacy risks posed by seemingly innocent sensors like accelerometers [Fitzpatrick 2010] illustrate the danger.

Resolving the tension between the fun and utility of running third-party mobile applications and the privacy risks they pose is a critical challenge for smartphone platforms. Mobile-phone operating systems currently provide only coarse-grained controls for regulating whether an application can *access* private information, but provide little insight into how private information is actually used. For example, if a user allows an application to access her location information, she has no way of knowing if the application will send her location to a location-based service, to advertisers, to the application developer, or to any other entity. As a result, users must blindly trust that applications will properly handle their private data.

This article describes *TaintDroid*, an extension to the Android mobile-phone platform that tracks the flow of privacy-sensitive data through third-party applications. TaintDroid assumes that downloaded, third-party applications are not trusted, and monitors—in realtime—how these applications access and manipulate users' personal data. Our primary goals are to detect when sensitive data leaves the system via untrusted applications and to facilitate analysis of applications by phone users or external security services [Lookout 2010; WhatApp 2010].

Analysis of applications' behavior requires sufficient contextual information about what data leaves a device and where it is sent. Thus, TaintDroid automatically labels data from privacy-sensitive sources and transitively applies labels as sensitive data propagates through program variables, files, and interprocess messages. When tainted data is transmitted over the network or otherwise leaves the system, TaintDroid logs the data's labels, the application responsible for transmitting the data, and the data's destination. Such realtime feedback gives users and security services greater insight into what mobile applications are doing and can potentially identify misbehaving applications.

To be practical, the performance overhead of the TaintDroid runtime must be minimal. Unlike existing solutions that rely on heavyweight whole-system emulation [Chow et al. 2004; Yin et al. 2007], we leveraged Android's virtualized architecture to integrate four granularities of taint propagation: variable-level, method-level, message-level, and file-level. Though the individual techniques are not new, our contributions lie in the integration of these techniques and in identifying an appropriate trade-off between performance and precision for resource-constrained smartphones. Experiments with our prototype for Android with JIT show that tracking incurs a runtime overhead of less than 32% for a CPU-bound microbenchmark. More importantly, interactive third-party applications can be monitored with negligible perceived latency.

We evaluated the precision of TaintDroid using 30 randomly selected, popular Android applications that use location, camera, or microphone data in April 2010. To provide a longitudinal perspective, in September 2012, we analyzed updated versions

of these same applications, of which only 20 still existed. In the 2010 study, TaintDroid revealed that 15 of the 30 applications reported users' locations to remote advertising servers. Seven applications collected the device ID and, in some cases, the phone number and the SIM card serial number. In all, two-thirds of the applications in our 2010 study used sensitive data suspiciously; so did a similar fraction of the tested applications in our 2012 study. Our findings demonstrate that TaintDroid can help expose potential misbehavior by third-party applications.

Like similar information-flow tracking systems [Chow et al. 2004; Yin et al. 2007], a fundamental limitation of TaintDroid is that it can be circumvented through leaks via implicit flows. The use of implicit flows to avoid taint detection is, in and of itself, an indicator of malicious intent, and may well be detectable through other techniques such as automated static code analysis [Denning and Denning 1977; Sabelfeld and Myers 2003] as we discuss in Section 8.

The rest of this article is organized as follows: Section 2 provides a high-level overview of TaintDroid, Section 3 describes background information on the Android platform, Section 4 describes the design of TaintDroid, Section 5 describes the taint sources tracked by TaintDroid, Section 6 presents results from our Android application study, Section 7 characterizes the performance of our prototype implementation, Section 8 discusses the limitations of our approach, Section 9 describes related work, and Section 10 summarizes our conclusions.

2. APPROACH OVERVIEW

We seek to design a framework that allows users to monitor how third-party smartphone applications handle their private data in realtime. Many smartphone applications are closed source and obfusticated, making static analysis infeasible. Even if source code is available (e.g., via decompilation [Enck et al. 2011]), runtime events and configuration often dictate information use—realtime monitoring can handle these environment-specific dependencies.

Monitoring network disclosure of privacy-sensitive information on smartphones presents several challenges.

— *Smartphones are resource constrained.* Resource limitations preclude the use of heavyweight information tracking systems like Panorama [Yin et al. 2007].
— *Third-party applications are entrusted with several types of privacy-sensitive information.* The monitoring system must distinguish multiple information types, which requires additional computation and storage.
— *Context-based privacy-sensitive information is dynamic and can be difficult to identify even when sent in the clear.* For example, geographic locations are pairs of floating point numbers that frequently change and are hard to predict.
— *Applications can share information.* Limiting the monitoring system to a single application does not account for flows via files and IPC between applications, including core system applications designed to disseminate privacy-sensitive information.

We use dynamic taint analysis [Clause et al. 2007; Nair et al. 2007; Qin et al. 2006; Yin et al. 2007] (also called "taint tracking") to monitor privacy-sensitive information on smartphones. Sensitive information is first identified at a *taint source*, where a *taint marking* indicating the information type is assigned. Dynamic taint analysis tracks how labeled data impacts other data in a way that might leak the original sensitive information. This tracking is often performed at the instruction level. Finally, the affected data is identified before it leaves the system at a *taint sink* (usually the network interface).

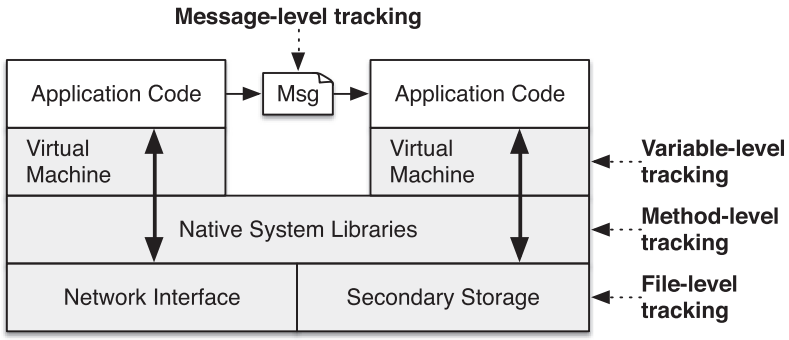

Fig. 1. Multilevel approach for performance-efficient taint tracking within a common smartphone architecture.

Existing taint tracking approaches have several limitations. First and foremost, approaches that rely on instruction-level dynamic taint analysis using whole-system emulation [Chow et al. 2004; Ho et al. 2006; Yin et al. 2007] incur high performance penalties. Instruction-level instrumentation incurs 2–20 times slowdown [Chow et al. 2004; Yin et al. 2007] in addition to the slowdown introduced by emulation, making it unsuitable for realtime analysis. Second, developing accurate taint propagation logic has proven challenging for the x86 instruction set [Newsome et al. 2009; Schwartz et al. 2010]. Implementations of instruction-level tracking can experience taint explosion if the stack pointer becomes falsely tainted [Slowinska and Bos 2009] and taint loss if complicated instructions such as CMPXCHG, REP MOV are not instrumented properly [Zhu et al. 2009]. While most smartphones use the ARM instruction set, similar false positives and false negatives could arise.

Figure 1 presents our approach to taint tracking on smartphones. We leverage architectural features of virtual-machine-based smartphones (e.g., Android, Windows Phone, and BlackBerry) to enable efficient, system-wide taint tracking using fine-grained labels with clear semantics. First, we instrument the VM interpreter to provide *variable-level tracking* within third-party application code. Using variable semantics provided by the interpreter provides valuable context for avoiding the taint explosion observed in the x86 instruction set. Additionally, by tracking variables, we maintain taint markings only for data and not code. Second, we use *message-level tracking* between applications. Tracking taint on messages instead of data within messages minimizes IPC overhead while extending the analysis system-wide. Third, for system-provided native libraries, we use *method-level tracking*. Here, we run native code without instrumentation and patch the taint propagation on return. These methods accompany the system and have known information-flow semantics. Finally, we use *file-level* tracking to ensure persistent information conservatively retains its taint markings.

To assign labels, we take advantage of the well-defined interfaces through which applications access sensitive data. For example, all information retrieved from GPS hardware is location sensitive, and all information retrieved from an address-book database is contact sensitive. This avoids relying on heuristics [Cox and Gilbert 2009] or manual specification [Zhu et al. 2009] for labels. We expand on information sources in Section 5.

In order to achieve this tracking at multiple granularities, our approach relies on the firmware's integrity. The taint tracking system's trusted computing base includes the virtual machine executing in userspace and any native system libraries loaded by the interpreted third-party application. However, this code is part of the firmware

and is therefore trusted. Applications can only escape the virtual machine by executing native methods. In our target platform (Android), we modified the native library loader to ensure that applications can only load native libraries from the firmware and not those downloaded by the application. To assess the impact of this design choice, we scanned the top 50 free applications in each category of the Android Market in mid-2010 and found less than 5% included a `.so` library file. A later study of over 200,000 Android applications had similar findings [Zhou et al. 2012]. Therefore, the number of applications affected by this design choice is relatively small. We leave dealing with third-party native libraries as future work.

In summary, we provide a novel, efficient, system-wide, multiple-marking taint tracking design by combining multiple granularities. While some techniques such as variable tracking within an interpreter have been previously proposed (see Section 9), to our knowledge, our approach is the first to extend such tracking system-wide. By choosing a multiple-granularity approach, we balance performance and precision.

3. BACKGROUND: ANDROID

Android is a Linux-based, open-source mobile-phone platform. Most core phone functionality is implemented as applications running on top of a customized middleware. Applications are written in Java and compiled to a custom bytecode format known as Dalvik EXecutable (DEX). Each application executes within its own Dalvik VM interpreter instance. Each instance executes as a unique UNIX user identity to isolate applications within the Linux platform. Applications communicate via the Binder IPC subsystem. The following discusses topics necessary to understand our tracking system.

Dalvik VM Interpreter. DEX is a register-based machine language, as opposed to Java bytecode, which is stack based. Each DEX method has its own predefined number of virtual registers (which we frequently refer to as simply "registers"). The Dalvik VM interpreter manages method registers with an internal execution-state stack; the current method's registers are always on the top stack frame. These registers loosely correspond to local variables in the Java method and store primitive types and object references. All computation occurs on registers; therefore, values must be loaded from and stored to class fields before and after use, respectively. Note that DEX uses class fields for all long-term storage, unlike hardware-register-based machine languages (e.g., x86) that store values in arbitrary memory locations. This additional context aids taint propagation in our design.

As of Android 2.3, the Dalvik VM contains a Just-In-Time compiler (JIT) to boost performance by translating bytecode to optimized native code at runtime.

Native Methods. The Android middleware provides access to native libraries for performance optimization and to support third-party libraries such as OpenGL and Webkit. Additionally, Android's Java implementation, which is based on Apache Harmony (harmony.apache.org), frequently uses system libraries to implement functionality (e.g., math routines). Native methods are written in C/C++ and expose functionality provided by the underlying Linux OS. They can also access Java internals and hence are included in our trusted computing base (Section 2).

Android contains two types of native methods: internal VM methods and JNI methods. The internal VM methods access interpreter-specific structures and APIs. JNI methods conform to Java native interface standards specifications [Liang 1999], which requires Dalvik to separate Java arguments into variables using a JNI call bridge. Conversely, internal VM methods must manually parse arguments from the interpreter's byte array of arguments. This difference impacts our design.

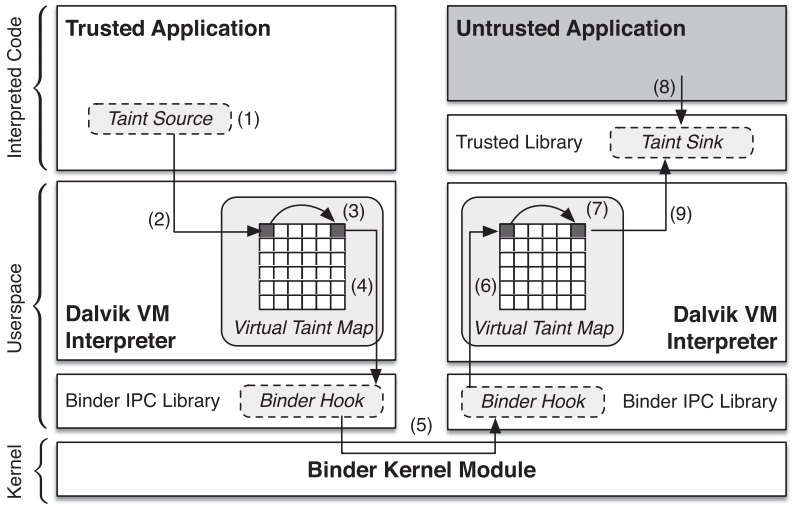

Fig. 2. TaintDroid architecture within Android.

Binder IPC. Android's IPC subsystem is called Binder. Fundamental to Binder are *parcels*, which serialize both active and standard data objects. The former includes references to Binder objects, which allows the framework to manage shared data objects between processes. A Binder kernel module passes parcels between processes.

4. TAINTDROID

TaintDroid is a realization of our multiple-granularity taint tracking approach within Android. TaintDroid uses variable-level tracking within the VM interpreter. Multiple taint markings are stored as one *taint tag*. When applications execute native methods, variable taint tags are patched on return. Finally, taint tags are assigned to parcels and propagated through Binder.

Figure 2 depicts TaintDroid's architecture. Information is tainted (1) in a trusted application with sufficient context (e.g., the location provider). The taint interface invokes a native method (2) that interfaces with the Dalvik VM interpreter, storing specified taint markings in the virtual taint map. The Dalvik VM propagates taint tags (3) according to dataflow rules as the trusted application uses the tainted information. Every interpreter instance simultaneously propagates taint tags. When the trusted application uses the tainted information in an IPC transaction, the modified binder library (4) ensures the parcel has a taint tag reflecting the combined taint markings of all contained data. The parcel is passed transparently through the kernel (5) and received by the remote untrusted application. Note that only the interpreted code is untrusted. The modified binder library retrieves the taint tag from the parcel and assigns it to all values read from it (6). The remote Dalvik VM instance propagates taint tags (7) identically for the untrusted application. When the untrusted application invokes a library specified as a taint sink (8), for example, network send, the library retrieves the taint tag for the data in question (9) and reports the event.

Implementing this architecture requires addressing several system challenges, including: (*a*) taint tag storage, (*b*) interpreted code taint propagation, (*c*) native code taint propagation, (*d*) IPC taint propagation, and (*e*) secondary storage taint propagation. The remainder of this section describes our design and concludes with a discussion of our integration of TaintDroid with the JIT version of the Dalvik VM introduced in Android version 2.3.

4.1. Taint Tag Storage

The choice of how to store taint tags influences performance and memory overhead. Dynamic taint tracking systems commonly store tags for every data byte or word [Chow et al. 2004; Yin et al. 2007]. Tracked memory is unstructured and without content semantics. Frequently taint tags are stored in nonadjacent shadow memory [Yin et al. 2007] and tag maps [Zhu et al. 2009]. TaintDroid uses variable semantics within the Dalvik interpreter. We store taint tags adjacent to variables in memory, providing spatial locality.

Dalvik has five variable types that require taint storage: method local variables, method arguments, class static fields, class instance fields, and arrays. In all cases, we store a 32-bit bitvector with each variable to encode the taint tag, allowing 32 different taint markings.

Method Local Variables. Temporary variables used for computation within a method are called "method local variables" and are the primary use of registers in the Dalvik interpreter. Registers contain both primitive type (i.e., scalar) values and object references. Registers are always 32 bits. For 64-bit types (e.g., *long* and *double*), Dalvik stores the value in two adjacent 32-bit registers and manages the registers separately. It is left to the bytecode instructions to determine if one or two registers are used for computation. The Dalvik interpreter uses a stack to manage the registers. On method invocation, Dalvik pushes a new stack frame, allocating space for its registers. During execution, registers are referenced by an index offset from the current frame pointer. For example, register $v0$ is $fp[0]$, register $v1$ is $fp[1]$, and so on. On method termination, the stack frame is popped, losing the register values.

TaintDroid stores 32-bit taint tags for each register (regardless of its current taint state) by allocating room for double the number of registers during the stack frame push. Taint tags are stored immediately after registers for efficient reference (as depicted in Figure 3). TaintDroid accounts for tag storage by adjusting the frame pointer index for each register vi to $fp[2 \cdot i]$ (a left bit shift), with the corresponding taint tag in $fp[2 \cdot i + 1]$. Note that for 64-bit scalars allocated as two adjacent 32-bit registers, the taint tag is duplicated and stored after each register for consistency with the Dalvik design.

Method Arguments. Dalvik uses registers to pass argument values to methods. A target method can be either interpreted or native. Before a method is invoked, temporary copies of the specified argument registers are pushed onto the stack. If the target method is interpreted, the new values become high-numbered local variable registers in the callee stack frame. If the target method is native, a pointer to the stack top is passed to the native method. The target native method is passed a pointer to a byte array from which it must parse 32- and 64-bit values in using its method signature.

This difference in handling method arguments impacts taint tag storage, as shown in Figure 3. Arguments for interpreted methods have interleaved taint tags for consistency with local variable taint storage. Native methods, on the other hand, expect a specific format in the received byte array of arguments, and interleaving taint tags would require modification of all native methods. However, not all native methods require taint tags for correct taint propagation (see Section 4.3). Therefore, we append the argument taint tags to maintain compatibility and avoid extra modifications. Finally, we allocate space for the taint tag of the return value for native methods. For interpreted methods, the 32 bits go unused.

Class Fields. Similar to Java, Dalvik has both static and instance class fields. Static fields store one value per class definition and are shared across all class instances, whereas instance fields store a different value for each class instance. Storage of taint

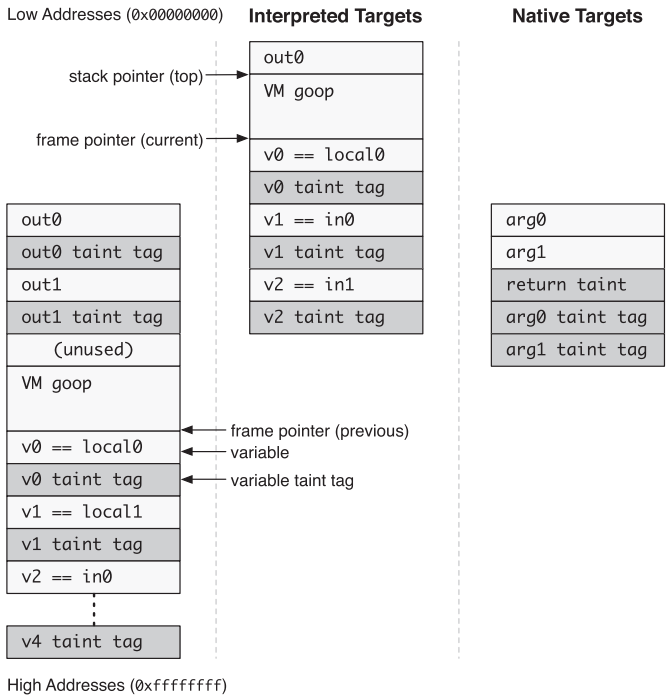

Fig. 3. Modified stack format. Taint tags are interleaved between registers for interpreted method targets and appended for native method arguments. Dark grayed boxes represent taint tags.

tags for static fields is straightforward: values are stored directly in a data structure managed by the interpreter. Instance fields require more careful instrumentation. The instance field data structure stores a byte-offset into a data object instance. Therefore we interleave taint tags with values in the instance data in the same fashion used for registers.[1]

Arrays. TaintDroid stores only one taint tag per array to minimize storage overhead. Per-value taint tag storage is severely inefficient for Java *String* objects, as all characters often have the same tag. Unfortunately, storing one taint tag per array may result in false positives during taint propagation. For example, if untainted variable u is stored into array A at index 0 ($A[0]$) and tainted variable t is stored into $A[1]$, then array A is tainted. Later, if variable v is assigned from $A[0]$, v will be tainted, even though u, which is the original variable assigned to $A[0]$, was untainted. Fortunately, Java frequently uses objects, and object references are infrequently tainted (see Section 4.2). Therefore, this coding practice leads to less false positives.

4.2. Interpreted Code Taint Propagation

The granularity and flow semantics of taint tracking influence both performance and precision. TaintDroid implements variable-level taint tracking within the Dalvik VM interpreter. Variables provide valuable semantics for taint propagation, distinguishing, for example, data pointers from scalar values. TaintDroid primarily tracks

[1]Readers familiar with Android may recognize that the optimized DEX (ODEX) format hardcodes field byte-offsets in the bytecode. However, the target device conventionally creates ODEX files on-demand, allowing TaintDroid to ensure compatibility.

primitive type variables (e.g., *int*, *float*, etc.); however, there are cases when object references must become tainted to ensure taint propagation operates correctly; this section addresses why these cases exist. However, first we present a formal logic for taint tracking in the Dalvik machine language.

4.2.1. Taint Propagation Logic. The DEX machine language used by Dalvik has a unique instruction set that requires a custom dataflow logic for taint propagation. We begin by defining taint markings, taint tags, variables, and taint propagation. We then present our logic rules for DEX.

Definition 1 (*Universe of Taint Markings* $\mathcal{L}$). Let each taint marking be a label l. We assume a fixed set of taint markings in any particular system. Example privacy-based taint markings include location, phone number, and microphone input. We define the universe of taint markings $\mathcal{L}$ to be the set of taint markings considered relevant for an application of TaintDroid.

Definition 2 (*Taint Tag*). A taint tag is a set of taint markings. A taint tag t is in the power set of $\mathcal{L}$, denoted $2^{\mathcal{L}}$, which includes $\emptyset$. Each variable has an associated tag that is dynamically updated based on logic rules.

Definition 3 (*Variable*). A variable is an instance of one of the five variable types described in Section 4.1 (method local variable, method argument, class static field, class instance field, and array). Variable types have different representations. The local and argument variables correspond to virtual registers, denoted v_x. Class field variables are denoted as f_x to indicate a field variable with class index x. f_x alone indicates a static field. Instance fields require an instance object and are denoted $v_y(f_x)$, where v_y is the instance object reference variable. Finally, $v_x[\cdot]$ denotes an array, where v_x is an array object reference variable.

Definition 4 (*Virtual Taint Map Function* $\tau(\cdot)$). Let v be a variable. $\tau(v)$ returns the taint tag t for variable v. $\tau(v)$ can also be used to assign a taint tag to a variable. Retrieval and assignment are distinguished by the position of $\tau(\cdot)$ with respect to the $\leftarrow$ symbol. When $\tau(v)$ appears on the right-hand side of $\leftarrow$, $\tau(v)$ retrieves the taint tag for v. When $\tau(v)$ appears on the left-hand side, $\tau(v)$ assigns the taint tag for v. For example, $\tau(v_1) \leftarrow \tau(v_2)$ copies the taint tag from variable v_2 to v_1.

Definitions 1–4 provide the primitives required to define runtime taint propagation for Dalvik VM. Table I captures the propagation logic. The table enumerates abstracted versions of the bytecode instructions specified in the DEX documentation. Register variables and class fields are referenced by v_X and f_X, respectively. R and E are the return and exception variables, respectively, maintained within the interpreter. A, B, and C are constants in the bytecode.

The taint propagation logic uses conservative dataflow semantics for constant, move, arithmetic, and logic instructions. Destination register values are always completely overwritten, therefore the taint tag is set explicitly for each instruction. Constant values are considered untainted and therefore do not contribute to the taint tag of the destination register. The interpreter maintains "hidden registers" for return and exception values. These registers require taint tag storage and corresponding propagation logic. The arithmetic and logic operations include unary negation, binary arithmetic, bit shifts, and bitwise AND and OR (abstracted as $\otimes$ in the table). Finally, the DEX bytecode does not require idioms to clear values (e.g., "`xor eax, eax`" in x86), therefore no special handling is required.

The array and class field access instructions (i.e., *put* and *get*) propagate taint tags almost identically to move instructions. However, as discussed in Section 4.2.2, there

Table I. DEX Taint Propagation Logic

Op Format	Op Semantics	Taint Propagation	Description
const-op v_A C	$v_A \leftarrow C$	$\tau(v_A) \leftarrow \emptyset$	Clear v_A taint
move-op v_A v_B	$v_A \leftarrow v_B$	$\tau(v_A) \leftarrow \tau(v_B)$	Set v_A taint to v_B taint
move-op-R v_A	$v_A \leftarrow R$	$\tau(v_A) \leftarrow \tau(R)$	Set v_A taint to return taint
return-op v_A	$R \leftarrow v_A$	$\tau(R) \leftarrow \tau(v_A)$	Set return taint ($\emptyset$ if void)
move-op-E v_A	$v_A \leftarrow E$	$\tau(v_A) \leftarrow \tau(E)$	Set v_A taint to exception taint
throw-op v_A	$E \leftarrow v_A$	$\tau(E) \leftarrow \tau(v_A)$	Set exception taint
unary-op v_A v_B	$v_A \leftarrow \otimes v_B$	$\tau(v_A) \leftarrow \tau(v_B)$	Set v_A taint to v_B taint
binary-op v_A v_B v_C	$v_A \leftarrow v_B \otimes v_C$	$\tau(v_A) \leftarrow \tau(v_B) \cup \tau(v_C)$	Set v_A taint to v_B taint $\cup$ v_C taint
binary-op v_A v_B	$v_A \leftarrow v_A \otimes v_B$	$\tau(v_A) \leftarrow \tau(v_A) \cup \tau(v_B)$	Update v_A taint with v_B taint
binary-op v_A v_B C	$v_A \leftarrow v_B \otimes C$	$\tau(v_A) \leftarrow \tau(v_B)$	Set v_A taint to v_B taint
aput-op v_A v_B v_C	$v_B[v_C] \leftarrow v_A$	$\tau(v_B[\cdot]) \leftarrow \tau(v_B[\cdot]) \cup \tau(v_A)$	Update array v_B taint with v_A taint
aget-op v_A v_B v_C	$v_A \leftarrow v_B[v_C]$	$\tau(v_A) \leftarrow \tau(v_B[\cdot]) \cup \tau(v_C)$	Set v_A taint to array and index taint
sput-op v_A f_B	$f_B \leftarrow v_A$	$\tau(f_B) \leftarrow \tau(v_A)$	Set field f_B taint to v_A taint
sget-op v_A f_B	$v_A \leftarrow f_B$	$\tau(v_A) \leftarrow \tau(f_B)$	Set v_A taint to field f_B taint
iput-op v_A v_B f_C	$v_B(f_C) \leftarrow v_A$	$\tau(v_B(f_C)) \leftarrow \tau(v_A)$	Set field f_C taint to v_A taint
iget-op v_A v_B f_C	$v_A \leftarrow v_B(f_C)$	$\tau(v_A) \leftarrow \tau(v_B(f_C)) \cup \tau(v_B)$	Set v_A taint to f_C and obj. ref. taint

Register variables and class fields are referenced by v_X and f_X, respectively. R and E are the return and exception variables maintained within the interpreter. A, B, and C are bytecode constants.

are several cases where the taint tag on the object reference must be included in the propagation logic.

DEX also defines several array- and class-related instructions not included in Table I (for brevity). The *array-length* instruction returns the length of an array, which we do not consider tainted. Note that some taint propagation logics taint this value to aid specific types of implicit flow propagation (e.g., Vogt et al. [2007]). The *new-array* and *fill-new-array* instructions allocate a new array with constant values. The *fill-data-array* copies constant values from the bytecode into an array. We assign a $\emptyset$ taint tag to the array if none of the original array content remains. Lastly, we do not consider the value returned from the *instance-of* class instruction to be tainted.

Finally, Table I does not show comparison instructions (*cmp-X*) that perform a comparison between registers and assign a destination register the value of 0, 1, or -1. Our implementation currently assigns the destination register a $\emptyset$ taint tag. Subsequent work [Gilbert et al. 2011] propagates taint tags from the argument registers to help define a "taint scope" for tracking specific types of implicit flows.

4.2.2. Tainting Object References. The propagation rules in Table I are straightforward with two exceptions. First, taint propagation logics commonly include the taint tag of the array index to handle translation tables (e.g., ASCII to UNICODE or character case conversion). For example, consider a translation table from lowercase to uppercase characters: if a tainted value "a" is used as an array index, the resulting "A" value should be tainted even though the "A" value in the array is not. Hence, the taint logic for *aget-op* uses both the array and array index taint. Second, when the array contains object references (e.g., an *Integer* array), the index taint tag is propagated to the object reference and not the object value. Since DEX provides object semantics to our propagation logic, we are able to include the object reference taint tag in the instance *get* (*iget-op*) rule.

The code listed in Figure 4 demonstrates a real instance of where object reference tainting is needed. Here, *valueOf()* returns an Integer object for a passed *int*. If the *int* argument is between -128 and 127, *valueOf()* returns reference to a statically defined

```
public static Integer valueOf(int i) {
  if (i < -128 || i > 127) {
    return new Integer(i); }
  return valueOfCache.CACHE [i+128];
}
static class valueOfCache {
  static final Integer[] CACHE = new Integer[256];
  static {
    for(int i=-128; i<=127; i++) {
      CACHE[i+128] = new Integer(i); } }
}
```

Fig. 4. Excerpt from Android's Integer class illustrating the need for object reference taint propagation.

Integer object. *valueOf()* is implicitly called for conversion to an object. Consider the following definition and use of a method *intProxy()*.

```
Object intProxy(int val) { return val; }
int out = (Integer) intProxy(tVal);
```

Consider the case where *tVal* is an *int* with value 1 and taint tag *TAG*. When *intProxy()* is passed *tVal*, *TAG* is propagated to *val*. When *intProxy()* returns *val*, it calls *Integer.valueOf()* to obtain an *Integer* instance corresponding to the scalar variable *val*. In this case, *Integer.valueOf()* returns a reference to the static Integer object with value 1. The *value* field (of the *Integer* class) in the object has taint tag of $\emptyset$; however, since the *aget-op* propagation rule includes the taint of the index register, the object reference has a taint tag of *TAG*. Therefore, only by including the object reference taint tag when the *value* field is read from the Integer (i.e., the *iget-op* propagation rule) will the correct taint tag of *TAG* be assigned to *out*.

4.3. Native Code Taint Propagation

Native code is not monitored by TaintDroid. We would like to have the same propagation semantics as interpreted code; therefore we define two *necessary postconditions*: (1) all accessed external variables (i.e., class fields referenced by native methods) are assigned taint tags according to the dataflow rules; and (2) the return value is assigned a taint tag according to the dataflow rules. TaintDroid achieves these postconditions through an assortment of manual instrumentation, method profiles, and heuristics depending on situational requirements.

4.3.1. Internal VM Methods. Internal VM methods are called directly by interpreted code, passing a pointer to an array of 32-bit register arguments and a pointer to a return value. The stack augmentation shown in Figure 3 provides access to taint tags for both Java arguments and the return value. Recall from Section 4.1 that TaintDroid appends the argument array with taint tag storage for each argument as well as the return value. We manually inspected Dalvik's internal VM methods and instrumented those requiring taint propagation. We also modified the interpreter to copy the return value taint tag from the argument array to the internally managed return value taint tag after the method terminates.

There are a relatively small number of internal VM methods which are infrequently added between versions. Only 11 internal VM methods were added between versions 1.5 and 2.1 (primarily for debugging and profiling). Between 2.1 and 2.3.4, 9 methods were removed and 10 methods were added (again methods were primarily for debugging and profiling). For the total 186 internal VM methods in Android version 2.3.4, only 5 required patching: the *System.arraycopy()* native method for copying array contents, and several native methods implementing Java reflection.

4.3.2. JNI Methods. The vast majority of Android's native methods use the Java Native Interface. JNI methods are invoked through the JNI call bridge, which is in and of itself an internal VM method. The call bridge parses Java arguments and assigns a return value based on the method's descriptor string. We patched the call bridge to provide taint propagation for all JNI methods using *method profiles*. A method profile is a list of (*from*, *to*) pairs indicating flows between variables, which may be method parameters, class variables, global variables, or return values. The current implementation of TaintDroid uses a manual definition of method profiles based on empirical findings. However, given the large volume of JNI methods, this task is much better suited for automated source-code analysis.

Instead of defining method profiles for every JNI method, we introduced a heuristic that provides conservative taint propagation for a large class of native methods that do not reference objects in their arguments, return value, or method body. The lack of referencing objects indicates information must flow from the method arguments to the return value. For example, the native *cos()* implementation returns the cosine of the passed argument. More generally, our heuristic assigns the union of the argument taint tags to the return value taint tag. Note this conservative calculation may cause false positives.

The heuristic has only false negatives for methods using objects. Objects allow information flows other than to the return value. Information may flow into an object directly or indirectly referenced by: (1) a method argument, (2) a field in the method's class, or (3) the return value. To expand coverage, we extend the heuristic to recognize object references to arrays and Java *String* objects when used as arguments and the return value. While this heuristic has both false positives and false negatives, we empirically found it effective in our experiments.

4.4. IPC Taint Propagation

Taint tags must propagate between applications when they exchange data. The tracking granularity affects performance and memory overhead. Therefore, to balance overhead and precision, our original implementation of TaintDroid [Enck et al. 2010] uses message-level taint tracking. While we found message-level taint tracking to be largely effective in practice (i.e., minimal false positives), we have also experimented with more precise byte-level tracking for IPC. We now discuss both variants.

Message-level IPC tracking. Recall that Android's IPC uses binder messages called *parcels*. Android constructs parcels in a userspace library that we modified to include taint tag information. For message-level IPC tracking, we added a taint tag variable to the parcel object. When new data is written to the parcel, we bitwise OR (i.e., union) the new data's tag with the existing tag and store the result. When the parcel is eventually transmitted over binder, the taint tag is appended to the data buffer, making the modifications transparent to the kernel. On the receiver side, the taint tag is immediately retrieved and automatically assigned to all variables read from the parcel.

Fine-grained IPC tracking. Message-level taint propagation for IPC can lead to false positives, because all data in the message shares the same taint tag. However, finer-grained tracking will incur a higher performance overhead. Since our original paper [Enck et al. 2010], we have explored the best way to include fine-grained tracking.

We decided to implement byte-level tracking. We could not simply use a shadow parcel, because it would increase IPC size by 400%. Instead, we chose a taint tag vector approach that is slightly slower, but significantly more memory efficient in the average case. The taint tag vector contains a start position and number of bytes for each tainted variable in the parcel. If two taint variables are adjacent, the entries, vector entries are combined. On the receiver side, our parcel library changes apply taint tags

automatically to variables based on the taint vector if any part of the variable's bytes are tainted.

During our experimentation, we observed false positives due to message-level tracking only when tracking the IMSI (see Section 8). However, byte-level tracking imposes a greater performance overhead. Section 7 compares the performance of the two approaches.

4.5. Secondary Storage Taint Propagation

Taint tags may be lost when data is written to a file. To address this, our design stores one taint tag per file. The taint tag is updated on file write and propagated to data buffers on file read. Taint tags are stored in the file system's extended attributes. For our original implementation of TaintDroid [Enck et al. 2010], we implemented extended attribute support for Android's host file system (YAFFS2). However, the YAFFS2 implementation in Android 2.3 includes xattr support, removing the need to modify the kernel. Additionally, by default Android formats SDCards as FAT, which does not support xattr-like storage. Therefore, we made minimal modifications to Android to support ext2-formatted SDcards. Interestingly, we found that some applications assume the SDcard can be used to share files between UIDs. Therefore we needed to explicitly set the permission mode to 666 and 777 when creating files and directories, respectively. These permissions are only applied to files and directories created under `/sdcard`. Finally, note that, as with arrays and IPC, storing one taint tag per file leads to false positives and limits the granularity of taint markings for databases (see Section 5). Alternatively, we could track taint tags at a finer granularity at the expense of added memory and performance overhead.

4.6. Taint Interface Library

Taint sources and sinks defined within the virtualized environment must communicate taint tags with the tracking system. We abstract the taint source and sink logic into a single taint interface library implemented as the *dalvik.system.Taint* Java class. The interface performs two functions: (1) adding taint markings to variables; and (2) retrieving taint markings from variables. The library only provides the ability to add and not set or clear taint tags, as such functionality could be used by untrusted Java code to remove taint markings.

Adding taint tags to arrays and strings via internal VM methods is straightforward, as both are stored in data objects. However, because scalar values are stored in the stack frame, this method cannot add the taint to the passed value. Instead, it returns a value that is tainted: `tainted = Taint.addTaintInt(foo, tag)`. Retrieving the tag of a variable is similar: `tag = Taint.getTaintInt(foo)`. Note that the stack storage does not pose complications for taint tag retrieval.

4.7. Integration with Dalvik JIT

With the release of Android 2.3, a just-in-time compiler (JIT) was added to the Dalvik VM to boost performance for bytecode execution [Cheng and Buzbee 2010]. JITs typically seek to improve performance by identifying frequently executed bytecode sequences and translating them to optimized native code at runtime. In the Dalvik VM, the interpreter profiles execution to identify candidate "hot" traces, which are subsequently compiled and optimized by the JIT. Within a trace, the JIT applies standard optimizations such as register promotion, redundant load/store elimination, and various loop optimizations. Once compiled and optimized, traces are stored in a per-process cache and reused when the same execution path is traversed again. JITs typically provide the greatest performance gains for compute-intensive workloads—the Dalvik VM

JIT achieves significant speedups of 2x to 5x for such workloads [Cheng and Buzbee 2010].

Because JIT has the potential to provide substantial performance gains over the standard interpreter, it is important for TaintDroid to support JIT to avoid imposing an unnecessary performance penalty for applications that benefit from it. Adding taint tracking to the JIT required modifying trace compilation, namely, the process by which a sequence of DEX bytecodes is translated to an intermediate representation (IR) amenable to optimization. A handler for each bytecode instruction generates a sequence of IR instructions implementing the bytecode's logic. We modified these handlers to insert the same taint propagation logic implemented in the Dalvik interpreter. Once a bytecode sequence and its corresponding taint propagation logic have been translated to IR form, the JIT proceeds with the normal steps of optimizing the trace and compiling it to ARM native code.

5. PRIVACY HOOK PLACEMENT

Using TaintDroid for privacy analysis requires identifying and instrumenting privacy-sensitive sources. Complex operating systems such as Android provide applications information in a variety of ways, such as direct access, as well as service interface. Each potential type of privacy-sensitive information must be studied carefully to determine the best method of defining the taint source. Taint sources can only add taint tags to memory for which TaintDroid provides tag storage. This section discusses how valuable taint sources and sinks can be implemented within these restrictions. We generalize such taint sources based on information characteristics.

Low-Bandwidth Sensors. A variety of privacy-sensitive information types are acquired through low-bandwidth sensors, for example, location and accelerometer. Such information often changes frequently and is simultaneously used by multiple applications. Therefore, it is common for a smartphone OS to multiplex access to low-bandwidth sensors using a manager. This sensor manager represents an ideal point for taint source hook placement. For our analysis, we placed hooks in Android's LocationManager and SensorManager services, which are part of our trusted computing base.

High-Bandwidth Sensors. Privacy-sensitive information sources such as the microphone and camera are high bandwidth. Each request from the sensor frequently returns a large amount of data that is only used by one application. Therefore, the smartphone OS may share sensor information via large data buffers, files, or both. When sensor information is shared via files, the file must be tainted with the appropriate tag. Due to flexible APIs, we placed hooks for both data buffer and file tainting for tracking microphone and camera information.

Information Databases. Shared information such as address books and SMS messages are often stored in file-based databases and accessed through Android's content provider programming abstraction. There are several possibilities for adding taint sources to content providers. Our original implementation [Enck et al. 2010] added the taint tag directly to the file storing the data, thereby tainting all data read from the file. The current implementation patches Android's *ContentResolver* and *CursorWrapper* classes to programmatically taint query response strings based on the content provider authority. For example, an address-book taint marking is added for the "*com.android.contacts*" authority. Similar rules are added for bookmarks, SMS, and MMS. Note that, while this file-level granularity was appropriate for these valuable information sources, others may exist for which files are too coarse grained. However, we have not yet encountered such sources.

Device Identifiers. Information that uniquely identifies the phone or the user is privacy sensitive. Not all personally identifiable information can be easily tainted, however, the phone contains several easily tainted identifiers: the phone number, SIM card identifiers (IMSI, ICC-ID), and device identifier (IMEI) are all accessed through well-defined APIs. We instrumented the APIs for the phone number, ICC-ID, and IMEI. An IMSI taint source has inherent limitations discussed in Section 8.

Network Taint Sink. Our privacy analysis identifies when tainted information is transmitted over the network interface. The VM interpreter-based approach requires the taint sink be placed within interpreted code. Hence, we instrumented the Java framework libraries at the point the native socket library is invoked.

6. APPLICATION STUDY

This section reports on two application studies that use TaintDroid to analyze how popular Android applications use privacy-sensitive user data. The first study was performed on 30 applications downloaded in April 2010 as part of our original TaintDroid paper [Enck et al. 2010]. To provide a longitudinal perspective, in September 2012 we downloaded updated versions of these same applications, of which only 20 still existed.

6.1. Experimental Setup

Our 2010 application study selected 30 applications as follows. We started with a snapshot of the 50 most popular free applications in Android Market in April 2010 (1,100 apps in total). We then narrowed the list by considering only those apps that had both Internet permission and at least one of location, camera, and audio permissions (358 apps). From this set, we randomly selected 30 applications for study (8.4% sample size) that span 12 categories. Table II enumerates these 30 apps. For our September 2012 study, we attempted to download updated versions of these 30 apps; however, 10 of the applications no longer existed. Furthermore, two of the updated apps depended on third-party native libraries and therefore failed to run in TaintDroid.

For both studies, we started each application, performed any initialization or registration required, and then manually exercised the offered functionality. We recorded system logs including detailed information from TaintDroid: tainted binder messages, tainted file output, and tainted network messages with the remote address. To verify our results, we also logged the network traffic using tcpdump on the WiFi interface and repeated experiments on multiple Nexus One phones running TaintDroid (based on Android 2.1 for the 2010 study and Android 2.3 for the 2012 study). The phones used for experiments had a valid SIM card; however, the SIM card was inactive, forcing all the packets to be transmitted via the WiFi interface. The packet trace was used only to verify the exposure of tainted data flagged by TaintDroid.

In addition to the network trace, we also noted whether applications acquired user consent (either explicit or implicit) for exporting sensitive information. This provides additional context for identifying possible privacy violations. For example, by selecting the "use my location" option in a weather application, the user implicitly consents to disclosing geographic coordinates to the weather server.

6.2. Findings

Table II shows the results of both the 2010 and 2012 study. The table classifies disclosures into implicit consent, explicit consent, and potential privacy violation categories based on the resolved host name corresponding to the network destination. We consider a disclosure to have implicit consent if it is clear that information will be sent to a host based on the performed actions. A disclosure has explicit consent if the user explicitly agrees to an EULA or terms of service upon starting the application. This

Table II. Information Sent Off Device in 2010 Study (left semicircle) and 2012 Study (right semicircle)

Application [package.name]	Location	IMEI	Ph. #	ICC-ID	IMSI	Contacts	Browser	Camera	Mic.	App List
3001 Wisdom Quotes Lite [com.xim.wq_lite]	●	◗								◗
Antivirus Free [com.antivirus]	●*	◖	◖	◖	◖		▯			
Astrid [com.timsu.astrid]	◖					◗*		▯		
Barcode Scanner [com.google.zxing.client.android]										
Blackjack [spr.casino]	◖									
Cestos Full [com.chickenbrickstudios.cestos_full]		◗								
The coupons App [thecouponsapp.coupon]	◖	◖	◖	◖	◖					
The directory for Germany [de.dastelefonbuch.android]	○									
Evernote [com.evernote]	◗*		◗			▯		○	○	
Hearts (Free) [com.bytesequencing.hearts_ads]	◖	◗								
Horoscope [fr.telemaque.horoscope]	●	●								
iXmat Barcode Scanner [com.ixellence.ixmat.android.community]										
Myspace [com.myspace.android]	◗	◗								
Solitare Free [com.mediafill.solitaire]	◖	◗								
Traffic Jam Free [com.jiuzhangtech.rushhour]	◖									
Trapster [com.trapster.android]	●*	◐								
The Weather Channel [com.weather.Weather]	◑									
Yellow Pages [com.avantar.yp]	◐	●								
Updated Version Crashes Due to Native Library										
Bump [com.bumptech.bumpga]	◖	●								
Layar [com.layar]	○	◖								
No longer in Android Market										
Babble Book [com.kalicinscy.babble]										
BBC News listen & tweet [daaps.media.bbc]	◖									
Children's ABC Animals (lite) [com.mob4.childrenabc.animals]	◖									
Knocking Live Video Beta [com.pointyheadsllc.knockingvideo]		◖								
Mabilo Ringtones [mabilo.ringtones]	◖									
Manga Browser [com.mangabrowser.main]	◖									
Movies and showtimes [com.stylem.movies]	◑									
Pro Basketball Scores [com.plusmo.probasketballscores]	◖									
Slide: Spongebob [com.mob4.slideme.qw.android.spongebob]	◖									
Wertago for Nightlife [com.wertago]	◖	◖†								

2010 study: ◑ = implicit consent, ◖ = explicit consent, ◖ = potential violation
2012 study: ▯ = implicit consent, ◗ = explicit consent, ◗ = potential violation
* Multiple uses result in both implicit consent and potential violation. † Sent the hash of the value.

requires the user to select "I agree" (or similar) in a dialog. Some applications displayed a link to a privacy policy for a short duration, or listed a link to a privacy policy at the bottom of the screen or in a settings menu. We did not consider these applications to have explicit consent. Note that if an app had both implicit and explicit consent, we indicate it as implicit consent. Finally, the remaining disclosures were considered potential privacy violations. In some cases, information had multiple disclosures wherein some uses had implicit consent and others were potential privacy violations (as indicated in the table).

Location Data to Advertisement Servers. In the 2010 study, 15 of the 30 applications exposed location data to third-party advertisement or analytics servers without

requiring implict or explicit user consent. Of these disclosures, 5 apps sent geocoordinates to ad.qwapi.com, 5 apps to admob.com, 2 apps to ads.mobclix.com (1 sent location both to admob.com and ads.mobclix.com), and 4 apps sent location to data.flurry.com. Of the 15 applications, only 2 presented an EULA on first run; however, neither EULA indicated this practice. Note that Table II shows 19 applications had privacy violations in the 2010 study. This is because 4 applications (Bump, Coupons, Trapster, and Yellow Pages) sent the location, but not to one of the aforementioned ad servers.

Exposure of location information occurred both in plaintext and in binary format. The latter highlights TaintDroid's advantages over simple pattern-based packet scanning. Applications sent location data in plaintext to admob.com, ad.qwapi.com, ads.mobclix.com (11 applications) and in binary format to FlurryAgent (four applications). The plaintext location exposure to AdMob occurred in the following HTTP GET string.

```
...&s=a14a4a93f1e4c68&..&t=062A1CB1D476DE85B717D9195A6722A9&
d%5Bcoord%5D=47.661227890000006%2C-122.31589477&...
```

Investigating the AdMob SDK revealed the `s=` parameter is an identifier unique to an app publisher, and the `coord=` parameter provides the geographic coordinates.

For FlurryAgent, we confirmed location exposure by the following sequence of events. First, a component named "FlurryAgent" registers with the location manager to receive location updates. Then, TaintDroid log messages show the application receiving a tainted parcel from the location manager. Finally, the application reports "sending report to http://data.flurry.com/aar.do" after receiving the tainted parcel.

Device Unique ID. Twenty out of the 30 applications downloaded in 2010 required permissions to read phone state and the Internet. In the 2010 study, TaintDroid detected that 9 of those 20 apps sent the IMEI to a network server. The IMEI uniquely identifies a specific mobile phone and is used to prevent a stolen handset from accessing the cellular network. Seven out of the 9 applications did not specify IMEI collection in the EULA (if even present). We found 2 of the 7 applications included the IMEI when transmitting the device's geographic coordinates to their content server, potentially repurposing the IMEI as a client ID.

In comparison, 2 of the 9 applications treated the IMEI with more care. Layar displayed a privacy statement that clearly indicated that the application collects the device ID. Wertago uses a hash of the IMEI instead of the number itself, which does not reveal the value itself. We verified this practice by comparing results from two different phones.

Phone Information. In addition to the IMEI, in the 2010 study we found that 2 applications also transmitted to their servers: (1) the device's phone number, (2) the IMSI, which is a unique 15-digit code used to identify an individual user on a GSM network, and (3) the ICC-ID number, a unique SIM card serial number. We verified messages were flagged correctly by inspecting the plaintext payload.[2] In neither case was the user informed that this information was sent off the phone.

This finding demonstrates that Android's coarse-grained access control provides insufficient protection against third-party applications seeking to collect sensitive data. Moreover, we found that one application sent the phone information every time the phone boots. While this application displays a terms of use on first use, it does not

[2]Because of the limitation of the IMSI taint source as discussed in Section 8, we disabled the IMSI taint source for experiments. Nonetheless, TaintDroid's flag of the ICC-ID and the phone number led us to find the IMSI contained in the same payload.

specify collection of this highly sensitive data. Surprisingly, this application transmits the phone data immediately after install, before first use.

Legitimate Flags. In the 2010 study, 37 exposures identified by TaintDroid were deemed clearly legitimate use. The flags resulted from four applications and the OS itself while using the Google Maps for Mobile (GMM) API. TaintDroid's logs indicate an HTTP request with the "User-Agent: GMM . . . " header, but a binary payload. Given that GMM functionality includes downloading maps based on geographic coordinates, it is obvious that TaintDroid correctly identified location information in the payload. Our manual inspection of each message along with the network packet trace confirmed that there were no false positives. We note that there is a possibility of false negatives that are difficult to verify with the lack of the source code of the third-party applications.

6.3. Longitudinal Study

The 2012 study is intended to provide insight into how application developers are reacting to the increased awareness of consumer privacy concerns. When our original study was published [Enck et al. 2010], relatively little public attention was given to privacy problems in smartphone apps. Since this study, there have been many news stories and academic works providing further investigation [Enck et al. 2011; Grace et al. 2012; Hornyack et al. 2011].

Permission Changes. Focusing on four permission types considered in the original study [Enck et al. 2010] (location, camera, microphone, phone state), we found the following. Hearts removed the location permission. Trapster removed the camera permission. MySpace also removed the camera permission, but it added the phone-state permission. Evernote and Solitare Free also added the phone-state permission and a substantial number of new permissions; however, Evernote's new permissions were likely due to new functionality, whereas the purpose for Solitaire Free's new permissions (get accounts, read contacts, get tasks) was less clear.

Changes in Information Disclosure. Table II contrasts the 2010 and 2012 studies. In the 2010 study, 19 applications had potential location-privacy violations, and we detected that 15 of those applications sent the location to an ad or analytics server. Of these 15 apps, 7 were no longer available in the Android Market. Of the remaining 8, 3 still sent location to an ad or analytics server (though not necessarily the same server). The other 5 either no longer exposed location or did not expose it during our experiment. Note that, as mentioned before, Hearts removed its location permission. The 2010 study identified Trapster as a potential location-privacy violator, but not to an ad server. In the 2012 study, we noted implicit consent to send location to a Trapster server, but not to send it to flurry.com. There were also two new potential location-privacy violators. The MySpace app sent location information to e.com during login. We observed implicit consent for Evernote to send location to evernote.com, but not to google-analytics.com.

There were also significant changes to IMEI disclosure. Of the original seven potential violators, only three were confirmed to still have that behavior. However, five other applications were observed to disclose the IMEI without implicit or explicit consent. Note that, while three apps added phone-state permissions, none of the apps dropped the phone-state permission. Therefore, the privacy-violating functionality may still exist, but was not exercised during our experiments (despite repeated trials). Conversely,

the observed new functionality may have also existed in the 2010 version, but was not exercised (except when the previous version did not have the phone-state permission).

Next, we make special note of Antivirus Free and The Coupons App. Both of these applications were identified as exposing multiple types of phone information in the 2010 study. Antivirus Free still contains the phone-state permission, and deeper investigation is required to determine if the behavior has truly changed. As for The Coupons App, we noted a logged Java error with a URL containing the IMEI, IMSI, phone number, and location, as well as an email address. Therefore, without the error, the information may have been disclosed and therefore detected by TaintDroid.

Finally, our improved and added taint sources revealed two additional potential privacy violations. First, we observed the 3001 Wisdom Quotes Lite application sending the list of installed applications to applovin.com. Second, we observed Astrid sending address-book entries to ajax.googleapis.com and analytics.localytics.com. However, this occurred when a contact was used in a task name, and appears an unintended consequence of using the contact for this purpose.

Explicit Consent. EULAs and privacy policies are becoming more common in applications; however, we observed very few instances of explicit consent. The Weather Channel application opened a dialog box to ask the user to use location information for advertisements. Trapster displays a terms of service informing the user that it will collect the device ID and IP address (but not location).

Whether or not the app acquired explicit consent was not always clear. For example, Evernote sends location and phone number in an encoded format to google-analytics.com. It has a link at the bottom of the account signup screen indicating that the user agrees to the privacy policy when creating a new account. While the link to the privacy policy does indicate using personal information and geographic region, it does not mention the phone number. Furthermore, the link to the privacy policy may be easy to overlook for most users.

Other applications with links to privacy policies were even less accessible, appearing only for a few seconds during application launch. For example, in the 2012 experiment, when Solitaire Free was started for the first time, a small dialog was displayed indicating the *applovin* library will access user data to enable social gaming/advertisements. The dialog contains a privacy policy URL in small text at the bottom and is only shown several seconds, and only on first use. The privacy policy indicates the library had access to address book, applications installed, device information, and location. However, since the policy was difficult to see and the user did not need to accept it, we do not classify the corresponding disclosure as explicit consent. In other cases, such as, Cestos, a URL is never displayed and the user must visit the developer's Web site to find the privacy policy.

6.4. Summary

Our study shows the effectiveness of the TaintDroid system in accurately tracking applications' use of privacy-sensitive data. While monitoring these applications, TaintDroid generated no false positives (with the exception of the IMSI taint source which we disabled for experiments; see Section 8). The flags raised by TaintDroid helped to identify potential privacy violations. While our sample set is relatively small (only 30 of hundreds of thousands of apps), the ways in which privacy-sensitive information is disclosed indicates significant privacy concerns for smartphone applications.

7. PERFORMANCE EVALUATION

In this section, we measure the overhead incurred by TaintDroid. Experiments were performed on a Google Nexus One (macrobenchmarks and Java microbenchmark) and

Table III. Macrobenchmark Results

	Android	TaintDroid
App Load Time	79 ms	91 ms
Address Book (create)	275 ms	309 ms
Address Book (read)	89 ms	98 ms
Phone Call	171 ms	197 ms
Take Picture	2913 ms	2938 ms

Nexus S (IPC microbenchmark) running Android OS version 2.3. Within the interpreted environment, TaintDroid incurs the same performance and memory overhead regardless of the existence of taint markings. Hence, we only need to ensure that file access includes appropriate taint tags.

7.1. Macrobenchmarks

During the application study, we anecdotally observed limited performance overhead. We hypothesize that this is because: (1) most applications are primarily in a "wait state," and (2) heavyweight operations (e.g., screen updates and Web page rendering) occur in unmonitored native libraries.

To gain further insight into perceived overhead, we devised five macrobenchmarks for high-level smartphone operations. Each experiment was measured 50 times and observed 95% confidence intervals at least an order of magnitude less than the mean. In each case, we excluded the first run to remove unrelated initialization costs. Experimental results are shown in Table III.

Application Load Time. The application load time measures from when Android's Activity Manager receives a command to start an activity component to the time the activity thread is displayed. This time includes application resolution by the Activity Manager, IPC, and graphical display. TaintDroid adds only 15% overhead, as the operation is dominated by native graphics libraries.

Address Book. We built a custom application to create, read, and delete entries for the phone's address book, exercising both file read and write. Create used three SQL transactions while read used two. The subsequent delete operation was lazy, returning in 0ms, and hence was excluded from our results. TaintDroid adds approximately 12% and 10% overhead for address-book entry creates and reads, respectively. Note that the user experiences less than 40ms overhead when creating or viewing a contact.

Phone Call. The phone call benchmark measured the time from pressing "dial" to the point at which the audio hardware was reconfigured to "in call" mode. TaintDroid adds less than 30ms per phone call setup ($\sim$15% overhead), which is significantly less than call setup in the network that takes on the order of seconds.

Take Picture. The picture benchmark measures from the time the user presses the "take picture" button until the preview display is reenabled. This measurement includes the time to capture a picture from the camera and save the file to the SDcard. TaintDroid had an overhead of less than 1%.

7.2. Java Microbenchmark

Figure 5 shows the execution-time results of a Java microbenchmark. We used an Android port of the standard CaffeineMark 3.0 [Pendragon Software Corporation 1997].

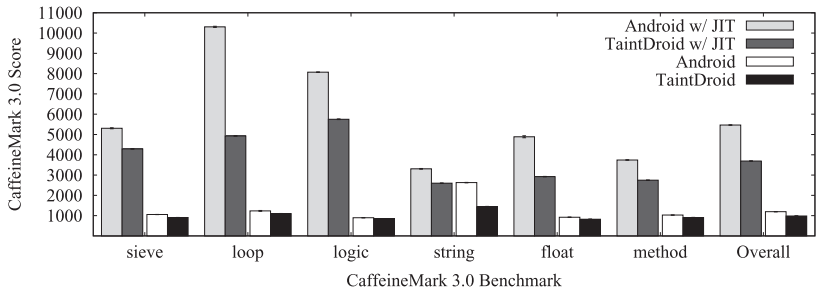

Fig. 5. Microbenchmark of Java overhead.

CaffeineMark uses an internal scoring metric that roughly corresponds to the number of Java instructions executed per second; as a result, scores are useful only for relative comparisons. The "overall" results indicate a cumulative score across individual benchmarks.

The unmodified Android system with JIT enabled had an average score of 5,466, while TaintDroid measured 3,693. TaintDroid has a 32% overhead with respect to the unmodified system in JIT mode. With JIT disabled, the scores for unmodified Android and TaintDroid were 1,194 and 983, respectively, indicating an 18% overhead for TaintDroid. The larger relative overhead for JIT mode is expected. A key factor in TaintDroid's low CPU overhead is the fact that executing the Dalvik bytecode instruction logic is only a small portion of the overall machine instructions required to execute the Dalvik VM. JIT mode reduces the time spent executing non-bytecode logic, and hence executing TaintDroid in JIT mode has a larger relative overhead.

7.3. IPC Microbenchmark

Our original TaintDroid implementation [Enck et al. 2010] only provided message-level IPC taint propagation. While our experimentation revealed only a limited number of false positives, we have since explored a finer-grained, byte-level taint tracking for IPC (see Section 4.4).

The IPC benchmark measures the overhead due to the parcel modifications for both *message-level* tracking and *fine-grained* byte-level tracking. For this experiment, we developed client and service applications that perform binder transactions as fast as possible. The client and service exchange messages of variable size: 1, 10, or 100 integer values. The experiment measures the time for the pair of applications to exchange messages 10,000 times. For each message size, we tainted zero, one, or all of the items contained in the parcel with unique taint tags.

Figure 6 shows the results of the IPC benchmark. TaintDroid with message-level tracking was about 35% slower than Android for single-item parcels and 230% slower for parcels containing 100 items. Since message-level tracking adds only four bytes to each IPC object, overhead due to data size is unlikely. A more likely cause of the overhead is the continual copying of taint tags as values are marshalled into and out of the parcel byte buffer. The increase in overhead as parcel size increases is likely due to a reduction of spatial locality when packing and unpacking parcels because of the taint tag appended at the end.

Fine-grained byte-level tracking is 50%–475% slower than Android. Byte-level tracking adds complexity to the parcel read and write routines. This results in a greater overhead than that of message-level tracking. For both message-level and byte-level tracking, the presence of tainted items in the message did not have a significant impact on the overhead — the greatest effect observed was an increase from 28.6 seconds

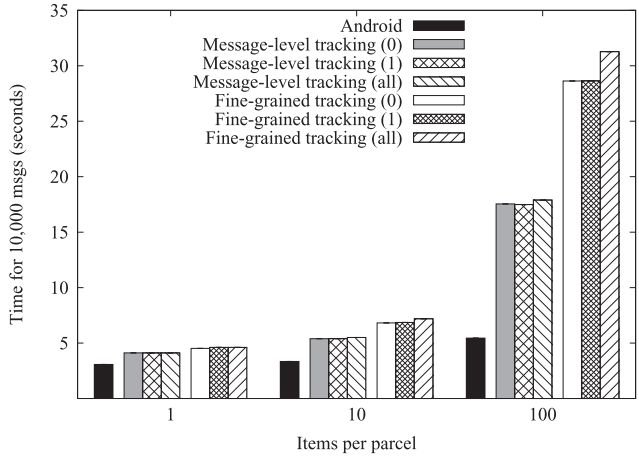

Fig. 6. IPC overhead microbenchmark.

for untainted 100-item parcels to 31.3 seconds for completely tainted parcels with byte-level tracking.

8. DISCUSSION

Approach Limitations. TaintDroid tracks only explicit dataflows in which a bytecode directly transfers information from its source objects to its destination objects. TaintDroid does not track implicit dataflows in which an app's control flow (e.g., conditional branching) or covert channels (e.g., volume control [Schlegel et al. 2011]) indirectly transfer sensitive information. Handling implicit flows is long-standing problem [Denning and Denning 1977] that requires carefully balancing undertainting, overtainting, and efficiency. TaintDroid is designed for minimal performance overhead, and Section 6 shows that TaintDroid tracks nonmalicious applications well enough to reveal a wide range of suspicious behavior. However, truly malicious apps can circumvent TaintDroid and exfiltrate privacy-sensitive information through implicit flows. Fully tracking implicit flows requires some form of static program analysis [Denning and Denning 1977; Myers 1999; Schwartz et al. 2010] to compute which objects' values are influenced by changes in control flow. One approach is to limit programmers' expressiveness so that the set of influenced objects is restricted [Myers 1999], but this approach cannot be applied to existing apps. Without programming-language restrictions, computing the set of influenced objects must be performed conservatively, leading to rampant overtainting and false alerts [Schwartz et al. 2010]. Several projects have explored selectively propagating taint based on the amount of sensitive information transferred by implicit flows [Cox et al. 2014; Kang et al. 2011; McCamant and Ernst 2008]. However, this approach is unlikely to be practical for the kinds programs and data types targeted by TaintDroid (e.g., location-based mobile apps). Finally, TaintDroid cannot track information flows that cross machine boundaries; once information leaves the phone, TaintDroid cannot detect when it returns in a network reply.

Implementation Limitations. Android uses the Apache Harmony [2011] implementation of Java with a few custom modifications. This implementation includes support for the *PlatformAddress* class, which contains a native address and is used by *DirectBuffer* objects. The file and network IO APIs include write and read "direct" variants that consume the native address from a *DirectBuffer*. TaintDroid does not currently track taint tags on *DirectBuffer* objects because the data is stored in opaque

native data structures. Currently, TaintDroid logs when a read or write "direct" variant is used, which anecdotally occurred with minimal frequency. Similar implementation limitations exist with the *sun.misc.Unsafe* class, which also operates on native addresses.

Taint Source Limitations. While TaintDroid is very effective for tracking sensitive information, it causes significant false positives when the tracked information contains configuration identifiers. For example, the IMSI numeric string consists of a Mobile Country Code (MCC), Mobile Network Code (MNC), and Mobile Station Identifier Number (MSIN) that are all tainted together.[3] Android uses the MCC and MNC extensively as configuration parameters when communicating other data. This causes all information in a parcel to become tainted, eventually resulting in an explosion of tainted information. Thus, for taint sources that contain configuration parameters, tainting individual variables within parcels is more appropriate. However, as our analysis results in Section 6 show, message-level taint tracking is effective for the majority of our taint sources, and byte-level tracking can be enabled if necessary.

Security Evaluation Limitations. TaintDroid trades off precision for performance in array, IPC, and file taint tracking. Due to coarse-grained tracking in these areas, TaintDroid might have false positivies. To check if positives are true or false, we may compare results of more fine-grained taint tracking systems with those of TaintDroid. However, even for the fine-grained taint tracking systems, we do not have ground truths for the applications we study. It is challenging to automatically infer ground truths to evaluate taint tracking systems; this is a great topic for future research.

9. RELATED WORK

Mobile-phone host security is a growing concern. OS-level protections such as Kirin [Enck et al. 2009], Saint [Ongtang et al. 2009], APEX [Nauman et al. 2010], CRePE [Conti et al. 2010], SEAndroid [Smalley and Craig 2013], and FlaskDroid [Bugiel et al. 2013] provide finer-grained controls to prevent access to sensitive information. Aurasium [Xu et al. 2012] and RetroSkeleton [Davis and Chen 2013] seek similar semantics by rewriting applications to include inline reference monitors. However, these approaches only prevent information from being accessed; once information enters the application, no additional mediation occurs. XManDroid [Bugiel et al. 2011a], TrustDroid [Bugiel et al. 2011b], and Aquifer [Nadkarni and Enck 2013] extend Android to track information flows at a coarse granularity between applications. TISSA [Zhou et al. 2011], MockDroid [Beresford et al. 2011], and AppFence [Hornyack et al. 2011] replace sensitive information with fake values. AppFence also extends TaintDroid [Enck et al. 2010] work with access control. CleanOS [Tang et al. 2012] modifies TaintDroid to enable secure deletion of information from application memory. From an integrity perspective, IPC Inspection [Felt et al. 2011] and Quire [Dietz et al. 2011] track Android intent messages through a chain of applications to prevent privilege escalation attacks.

Decentralized information-flow control (DIFC) enhanced operating systems such as Asbestos [Vandebogart et al. 2007] and HiStar [Zeldovich et al. 2006] label processes and enforce access control based on Denning's lattice model for information-flow security [Denning 1976]. Flume [Krohn et al. 2007] provides similar enhancements for legacy OS abstractions. DEFCon [Migliavacca et al. 2010] uses a logic similar to these

[3]Regardless of the string separation, the MCC and MNC are identifiers that warrant taint sources.

DIFC OSes, but focuses on events and modifies a Java runtime with lightweight isolation. Related to these system-level approaches, PRECIP [Wang et al. 2008] labels both processes and shared kernel objects such as the clipboard and display buffer. However, these process-level information-flow models are coarse grained and cannot track sensitive information *within* untrusted applications.

Tools that analyze applications for privacy-sensitive information leaks include Privacy Oracle [Jung et al. 2008] and TightLip [Yumerefendi et al. 2007]. These tools investigate applications while treating them as a black box, thus enabling analysis of off-the-shelf applications. However, this black-box analysis tool becomes ineffective when applications use encryption prior to releasing sensitive information.

Language-based information-flow security [Sabelfeld and Myers 2003] extends existing programming languages by labeling variables with security attributes. Compilers use the security labels to generate security proofs, such as Jif [Myers 1999; Myers and Liskov 2000] and SLam [Heintze and Riecke 1998]. Laminar [Roy et al. 2009] provides DIFC guarantees based on programmer-defined security regions. However, these languages require careful development and are often incompatible with legacy software designs [Hicks et al. 2006].

Dynamic taint analysis provides information tracking for legacy programs. The approach has been used to enhance system integrity (e.g., defend against software attacks [Clause et al. 2007; Newsome and Song 2005; Qin et al. 2006]) and confidentiality (e.g., discover privacy exposure [Egele et al. 2007; Yin et al. 2007; Zhu et al. 2009]), as well as to track Internet worms [Costa et al. 2005]. Dynamic tracking approaches range from whole-system analysis using hardware extensions [Crandall and Chong 2004; Suh et al. 2004; Vachharajani et al. 2004] and emulation environments [Chow et al. 2004; Yin et al. 2007] to per-process tracking using dynamic binary translation (DBT) [Cheng et al. 2006; Clause et al. 2007; Qin et al. 2006; Zhu et al. 2009]. The performance and memory overhead associated with dynamic tracking has resulted in an array of optimizations, including optimizing context switches [Qin et al. 2006], on-demand tracking [Ho et al. 2006] based on hypervisor introspection, and function summaries for code with known information-flow properties [Zhu et al. 2009]. If source code is available, significant performance improvements can be achieved by automatically instrumenting legacy programs with dynamic tracking functionality [Lam and cker Chiueh 2006; Xu et al. 2006]. Automatic instrumentation has also been performed on x86 binaries [Saxena et al. 2008], providing a compromise between source-code translation and DBT. TaintDroid's design was inspired by these prior works, but addressed different challenges unique to mobile phones. To our knowledge, TaintDroid is the first taint tracking system for a mobile phone and is the first dynamic taint analysis system to achieve practical system-wide analysis through the integration of tracking multiple data object granularities.

DroidScope [Yan and Yin 2012] is a recent system that implements taint tracking in the Android emulation layer for offline security analysis. DroidScope runs the analysis outside the smartphone software stack and can analyze kernel-level attacks. However, the system is not for realtime monitoring and cannot cover subtleties from real devices.

Finally, dynamic taint analysis has been applied to virtual machines and interpreters. Haldar et al. [2005] instrument the Java String class with taint tracking to prevent SQL injection attacks. WASP [Halfond et al. 2008] uses positive tainting of individual characters to ensure the SQL query contains only high-integrity substrings. Chandra and Franz [2007] propose fine-grained information-flow tracking within the JVM and instrument Java bytecode to aid control-flow analysis. Similarly, Nair et al. [2007] instrument the Kaffe JVM. Vogt et al. [2007] instrument a Javascript interpreter to prevent cross-site scripting attacks. Xu et al. [2006] automatically instrument the PHP interpreter source code with dynamic information tracking to prevent

SQL injection attacks. The Resin [Yip et al. 2009] environment for PHP and Python uses dataflow tracking to prevent an assortment of Web application attacks. When data leaves the interpreted environment, Resin implements filters for files and SQL databases to serialize and deserialize objects and policy with byte-level granularity. TaintDroid's interpreted code taint propagation bears similarity to some of these works. However, TaintDroid implements system-wide information-flow tracking, seamlessly connecting interpreter taint tracking with OS sharing mechanisms.

10. CONCLUSIONS

We have presented TaintDroid, an efficient, system-wide information-flow tracking tool that can simultaneously track multiple sources of sensitive data. TaintDroid achieves efficiency, a 32% performance overhead on a CPU-bound microbenchmark, by integrating four granularities of taint propagation (variable-level, message-level, method-level, and file-level).

We used our TaintDroid implementation to study the behavior of 30 popular third-party applications. Our 2010 study revealed that two-thirds of the applications in our study exhibit suspicious handling of sensitive data, and that 15 of the 30 applications reported users' locations to remote advertising servers. A similar fraction of the tested applications in our 2012 study also potentially misused users' sensitive data. Our findings demonstrate the effectiveness and value of enhancing smartphone platforms with TaintDroid.

TaintDroid is an ongoing effort that has been actively used by the systems security community. TaintDroid is available for Android version 2.1, version 2.3 (and adding JIT support), version 4.1, and version 4.3. Information for downloading and building TaintDroid can be found at http://appanalysis.org.

ACKNOWLEDGMENTS

We would like to thank Intel Labs, Berkeley and Seattle for its support and feedback during the design and prototype implementation of this work. We thank Jayanth Kannon, Stuart Schechter, and Ben Greenstein for their feedback during the writing of this article. We also thank Kevin Butler, Stephen McLaughlin, Machigar Ongtang, and the SIIS lab as a whole for their helpful comments.

REFERENCES

Alastair R. Beresford, Andrew Rice, Nicholas Skehin, and Ripduman Sohan. 2011. MockDroid: Trading privacy for application functionality on smartphones. In *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications (HotMobile'11)*.

Apache Harmony. 2011. Apache harmony – Open source Java platform. http://harmony.apache.org.

Apple, Inc. 2013. Apples app store downloads top three billion. http://www.apple.com/pr/library/2013/01/07App-Store-Tops-40-Billion-Downloads-with-Almost-Half-in-2012.html.

Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, and Ahmad-Reza Sadeghi. 2011a. Xmandroid: A new android evolution to mitigate privilege escalation attacks. Tech. rep. TR-2011-04, Center for Advanced Security Research Darmstadt, Technische Universitat Darmstadt, Darmstadt, Germany.

Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Stephan Heuser, Ahmad-Reza Sadeghi, and Bhargava Shastry. 2011b. Practical and lightweight domain isolation on android. In *Proceedings of the ACM Workshop on Security and Privacy in Mobile Devices (SPSM'11)*.

Sven Bugiel, Stephan Heuser, and Ahmad-Reza Sadeghi. 2013. Flexible and fine-grained mandatory access control on android for diverse security and privacy policies. In *Proceedings of the USENIX Security Symposium*.

Deepak Chandra and Michael Franz. 2007. Fine-grained information flow analysis and enforcement in a Java virtual machine. In *Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC'07)*.

Ben Cheng and Bill Buzbee. 2010. A jit compiler for androids dalvik vm. http://dl.google.com/googleio/2010/android-jit-compiler-androids-dalvik-vm.pdf.

Winnie Cheng, Qin Zhao, Bei Yu, and Scott Hiroshige. 2006. TaintTrace: Efficient flow tracing with dynamic binary rewriting. In *Proceedings of the IEEE Symposium on Computers and Communications (ISCC'06)*. 749–754.

Jim Chow, Ben Pfaff, Tal Garfinkel, Kevin Christopher, and Mendel Rosenblum. 2004. Understanding data lifetime via whole system simulation. In *Proceedings of the 13th USENIX Security Symposium*.

James Clause, Wanchun Li, and Alessandro Orso. 2007. Dytan: A generic dynamic taint analysis framework. In *Proceedings of the International Symposium on Software Testing and Analysis*. 196–206.

Mauro Conti, Vu Thien Nga Nguyen, and Bruno Crispo. 2010. CRePE: Context-related policy enforcement for android. In *Proceedings of the 13th Information Security Conference (ISC'10)*.

Manuel Costa, Jon Crowcroft, Miguel Castro, Antony Rowstron, Lidong Zhou, Lintao Zhang, and Paul Barham. 2005. Vigilante: End-to-end containment of internet worms. In *Proceedings of the ACM Symposium on Operating Systems Principles*. 133–147.

Landon P. Cox and Peter Gilbert. 2009. Redflag: Reducing inadvertent leaks by personal machines. Tech. rep. TR-2009-02, Duke University.

Landon P. Cox, Peter Gilbert, Geoffrey Lawler, Valentin Pistol, Ali Razeen, Bi Wu, and Sai Cheemalapati. 2014. Spandex: Secure password tracking for android. Tech. rep. TR-2014-01, Duke University.

Jedidiah R. Crandall and Frederic T. Chong. 2004. Minos: Control data attack prevention orthogonal to memory model. In *Proceedings of the International Symposium on Microarchitecture*. 221–232.

Chris Davies. 2009. iPhone spyware debated as app library "phones home". http://www.slashgear.com/iphone-spyware-debated-as-app-library-phones-home-1752491/.

Benjamin Davis and Hao Chen. 2013. RetroSkeleton: Retrofitting android apps. In *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys'13)*. 181–192.

Dorothy E. Denning. 1976. A lattice model of secure information flow. *Comm. ACM 19*, 5, 236–243.

Dorothy E. Denning and Peter J. Denning. 1977. Certification of programs for secure information flow. *Comm. ACM 20*, 7.

Michael Dietz, Shashi Shekhar, Yuliy Pisetsky, Anhei Shu, and Dan S. Wallach. 2011. Quire: Lightweight provenance for smart phone operating systems. In *Proceedings of the 20th USENIX Security Symposium*.

Manuel Egele, Christopher Kruegel, Engin Kirda, Heng Yin, and Dawn Song. 2007. Dyanmic spyware analysis. In *Proceedings of the USENIX Annual Technical Conference*. 233–246.

William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick Mcdaniel, and Anmol N. Sheth. 2010. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI'10)*.

William Enck, Damien Octeau, Patrick Mcdaniel, and Swarat Chaudhuri. 2011. A study of android application security. In *Proceedings of the 20th USENIX Security Symposium*.

William Enck, Machigar Ongtang, and Patrick Mcdaniel. 2009. On lightweight mobile phone application certification. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS'09)*.

Adrienne Porter Felt, Helen J. Wang, Alexander Moshchuk, Steven Hanna, and Erika Chin. 2011. Permission re-delegation: Attacks and defenses. In *Proceedings of the 20th USENIX Security Symposium*.

Michael Fitzpatrick. 2010. Mobile that allows bosses to snoop on staff developed. BBC News. http://news.bbc.co.uk/2/hi/technology/8559683.stm.

Peter Gilbert, Byung-Gon Chun, Landon P. Cox, and Jaeyeon Jung. 2011. Vision: Automated security validation of mobile apps at app markets. In *Proceedings of the International Workshop on Mobile Cloud Computing and Services (MCS'11)*.

Michael Grace, Wu Zhou, Xuxian Jiang, and Ahmad-Reza Sadeghi. 2012. Unsafe exposure analysis of mobile in-app advertisements. In *Proceedings of the ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec'12)*.

Vivek Haldar, Deepak Chandra, and Michael Franz. 2005. Dynamic taint propagation for Java. In *Proceedings of the 21st Annual Computer Security Applications Conference (ACSAC'05)*. 303–311.

William G. J. Halfond, Allesandro Orso, and Panagiotis Manolios. 2008. WASP: Protecting web applications using positive tainting and syntax-aware evaluation. *IEEE Trans. Softw. Engin. 34,* 1, 65–81.

Nevin Heintze and Jon G. Riecke. 1998. The slam calculus: Programming with secrecy and integrity. In *Proceedings of the Symposium on Principles of Programming Languages (POPL'98)*. 365–377.

Boniface Hicks, Kiyan Ahmadizadeh, and Patrick Mcdaniel. 2006. Understanding practical application development in security-typed languages. In *Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC'06)*. 153–164.

Alex Ho, Michael Fetterman, Christopher Clark, Andrew Warfield, and Steven Hand. 2006. Practical taint-based protection using demand emulation. In *Proceedings of the European Conference on Computer Systems (EuroSys'06)*. 29–41.

Peter Hornyack, Seungyeop Han, Jaeyeon Jung, Stuart Schechter, and David Wetherall. 2011. These aren't the droids you're looking for: Retrofitting android to protect data from imperious applications. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS'11)*.

Jaeyeon Jung, Anmol Sheth, Ben Greenstein, David Wetherall, Gabriel Maganis, and Tadayoshi Kohno. 2008. Privacy oracle: A system for finding application leaks with black box differential testing. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*. 279–288.

Min Gyung Kang, Stephen Mccamant, Pongsin Poosankam, and Dawn Song. 2011. DTA++: Dynamic taint analysis with targeted control-flow propagation. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'11)*.

Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. 2007. Information flow control for standard os abstractions. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP'07)*. 321–334.

Lap Chung Lam and Tzicker Chiueh. 2006. A general dynamic information flow tracking framework for security applications. In *Proceedings of the Annual Computer Security Applications Conference (AC-SAC'06)*. 463–472.

Sheng Liang. 1999. *Java Native Interface: Programmer's Guide and Specification*. Prentice Hall PTR.

Lookout. 2010. Introducing the app genome project. http://blog.mylookout.com/2010/07/introducing-the-app-genome-project/.

Stephen Mccamant and Michael D. Ernst. 2008. Quantitative information flow as network flow capacity. *SIGPLAN Not. 43*, 6, 193–205.

Matteo Migliavacca, Ioannis Papagiannis, David M. Eyers, Brian Shand, Jean Bacon, and Peter Pietzuch. 2010. DEFCon: High-performance event processing with information security. In *Proceedings of the USENIX Annual Technical Conference*.

Dan Moren. 2009. Retrievable iphone numbers mean potential privacy issues. http://www.macworld.com/article/143047/2009/09/phone_hole.html.

Andrew C. Myers. 1999. JFlow: Practical mostly-static information flow control. In *Proceedings of the ACM Symposium on Principles of Programming Langauges (POPL'99)*.

Andrew C. Myers and Barbara Liskov. 2000. Protecting privacy using the decentralized label model. *ACM Trans. Softw. Engin. Methodol. 9,* 4, 410–442.

Adwait Nadkarni and William Enck. 2013. Preventing accidental data disclosure in modern operating systems. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS'13)*.

Srijith K. Nair, Patrick N. D. Simpson, Bruno Crispo, and Andrew S. Tanenbaum. 2007. A virtual machine based information flow control system for policy enforcement. In *Proceedings of the 1st International Workshop on Run Time Enforcement for Mobile and Distributed Systems (REM'07)*.

Mohammad Nauman, Sohail Khan, and Xinwen Zhang. 2010. Apex: Extending android permission model and enforcement with user-defined runtime constraints. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security (ASICCAS'10)*. 328–332.

James Newsome, Stephen Mccamant, and Dawn Song. 2009. Measuring channel capacity to distinguish undue influence. In *Proceedings of the 4th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS'09)*. 73–85.

James Newsome and Dawn Song. 2005. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the 12th Network and Distributed System Security Symposium (NDSS'05)*.

Machigar Ongtang, Stephen Mclaughlin, William Enck, and Patrick Mcdaniel. 2009. Semantically rich application-centric security in android. In *Proceedings of the 25th Annual Computer Security Applications Conference (ACSAC'09)*.

Pendragon Software Corporation. 1997. CaffeineMark 3.0. http://www.benchmarkhq.ru/cm30/.

Feng Qin, Chen Wang, Zhenmin Li, Ho-Seop Kim, Yuanyuan Zhou, and Youfeng Wu. 2006. LIFT: A low-overhead practical information flow tracking system for detecting security attacks. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*. 135–148.

Indrajit Roy, Donald E. Porter, Michael D. Bond, Kathryn S. Mckinley, and Emmettwitchel. 2009. Laminar: Practical fine-grained decentralized information flow control. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'09)*. 63–74.

Andrei Sabelfeld and Andrew C. Myers. 2003. Language-based information-flow security. *IEEE J. Selected Areas Comm. 21*, 1, 5–19.

Prateek Saxena, R. Sekar, and Varun Puranik. 2008. Efficient fine-grained binary instrumentation with applications to taint-tracking. In *Proceedings of the IEEE/ACM Symposium on Code Generation and Optimization (CGO'08)*. 74–83.

Roman Schlegel, Kehuan Zhang, Xiao-Yong Zhou, Mehool Intwala, Apu Kapadia, and Xiao Feng Wang. 2011. Soundcomber: A stealthy and context-aware sound trojan for smartphones. In *Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS'11)*.

Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. 2010. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Proceedings of the IEEE Symposium on Security and Privacy*.

Asia Slowinska and Herbert Bos. 2009. Pointless tainting? Evaluating the practicality of pointer tainting. In *Proceedings of the European Conference on Computer Systems (EuroSys'09)*. 61–74.

Stephen Smalley and Robert Craig. 2013. Security enhanced (se) android: Bringing flexible MAC to android. In *Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS'13)*.

G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. 2004. Secure program execution via dynamic information flow tracking. In *Proceedings of the Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'04)*. 85–96.

Yang Tang, Phillip Ames, Sravan Bhamidipati, Ashish Bijlani, Roxana Geambasu, and Nikhil Sarda. 2012. CleanOS: Limiting mobile data exposure with idle eviction. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI'12)*.

Neil Vachharajani, Matthew J. Bridges, Jonathan Chang, Ram Rangan, Guilherme Ottoni, Jason A. Blome, George A. Reis, Manish Vachharajani, and David I. August. 2004. RIFLE: An architectural framework for user-centric information-flow security. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*. 243–254.

Steve Vandebogart, Petros Efstathopoulos, Eddie Kohler, Maxwell Krohn, Cliff Frey, David Ziegler, Frans Kaashoek, Robert Morris, and David Mazieres. 2007. Labels and event processes in the asbestos operating system. *ACM Trans. Comput. Syst. 25*, 4.

Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. 2007. Cross-site scripting prevention with dynamic data tainting and static analysis. In *Proceedings of the 14th Network and Distributed System Security Symposium*.

Xiao Feng Wang, Zhuowei Li, Ninghui Li, and Jong Youl Choi. 2008. PRECIP: Towards practical and retrofittable confidential information protection. In *Proceedings of 15th Network and Distributed System Security Symposium (NDSS'08)*.

Whatapp. 2010. WhatApp. http://www.whatapp.org.

Rubin Xu, Hassen Saidi, and Ross Anderson. 2012. Aurasium: Practical policy enforcement for android applications. In *Proceedings of the USENIX Security Symposium*.

Wei Xu, Sandeep Bhatkar, and R. Sekar. 2006. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *Proceedings of the USENIX Security Symposium*. 121–136.

Lok Kwong Yan and Heng Yin. 2012. DroidScope: Seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. In *Proceedings of the USENIX Security Symposium*.

Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. 2007. Panorama: Capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*. 116–127.

Alexander Yip, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. 2009. Improving application security with data flow assertions. In *Proceedings of the ACM Symposium on Operating Systems Principles*.

Aydan R. Yumerefendi, Benjamin Mickle, and Landon P. Cox. 2007. TightLip: Keeping applications from pilling the beans. In *Proceedings of the 4th USENIX Symposium on Network Systems Design and Implementation (NSDI'07)*. 159–172.

Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. 2006. Making information flow explicit in Histar. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI'06)*. 263–278.

Yajin Zhou, Zhi Wang, Wu Zhou, and Xuxian Jiang. 2012. Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. In *Proceedings of the Network and Distributed System Security Symposium*.

Yajin Zhou, Xinwen Zhang, Xuxian Jiang, and Vincent W. Freeh. 2011. Taming information-stealing smartphone applications (on android). In *Proceedings of the International Conference on Trust and Trustworthy Computing (TRUST'11)*.

David Zhu, Jaeyeon Jung, Dawn Song, Tadayoshi Kohno, and David Wetherall. 2009. Privacy scope: A precise information flow tracking system for finding application leaks. Tech. rep. EECS-2009-145, Department of Computer Science, UC Berkeley, CA.

Received April 2013; revised February 2014; accepted April 2014