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

William Enck
*The Pennsylvania State University*

Peter Gilbert
*Duke University*

Byung-gon Chun
*Intel Labs*

Landon P. Cox
*Duke University*

Jaeyeon Jung
*Intel Labs*

Patrick McDaniel
*The Pennsylvania State University*

Anmol N. Sheth
*Intel Labs*

## Abstract

Today's smartphone operating systems fail to provide users with adequate control and visibility into how third-party applications use their private data. We present TaintDroid, an efficient, system-wide dynamic taint tracking and analysis system for the popular Android platform that can simultaneously track multiple sources of sensitive data. TaintDroid's efficiency to perform real-time analysis stems from its novel system design that leverages the mobile platform's virtualized system architecture. TaintDroid incurs only 14% performance overhead on a CPU-bound micro-benchmark with little, if any, perceivable overhead when running third-party applications. We use TaintDroid to study the behavior of 30 popular third-party Android applications and find several instances of misuse of users' private information. We believe that TaintDroid is the first working prototype demonstrating that dynamic taint tracking and analysis provides informed use of third-party applications in existing smartphone operating systems.

## 1 Introduction

A key feature of modern smartphone platforms is centralized services for downloading third-party applications. The convenience to users and developers of such "app stores" has helped make mobile devices more fun and useful, and has led to an explosion of development. Apple's App Store alone served nearly two billion applications after only one year [4]. Many of these applications combine data from remote cloud services with information from local hardware sensors such as GPS, camera, microphone, and accelerometer and other sources. Applications often have legitimate reasons for accessing this privacy sensitive data, but users would also like some assurance that their data is being used properly. Users' unease is justified as some developers have been found to relay private information back to the cloud [32, 12], and even sensors as seemingly innocent as accelerometers can pose privacy risks [18].

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. The lack of transparency forces users to blindly trust that applications will handle private data properly once they are installed. For example, a user may wish to allow an application to access her location information so that she can participate in a location-based service, but in granting this access she must also trust that the application will not forward her location information and unique identifiers to advertising servers.

We present *TaintDroid*, an extension to the Android mobile-phone platform that tracks the flow of privacy sensitive data through third-party applications. TaintDroid automatically categorizes privacy-sensitive data sources and labels accordingly when applications obtain information from these sources. The runtime environment performs system-wide tracking of variables, files, and interprocess messages that propagate these data. When tainted data are transmitted over the network, or otherwise leave the system, TaintDroid identifies the categories of information with which they are tainted, the application responsible for transmitting the data, and the destination to which the data are being sent. Such realtime feedback generated by TaintDroid can give users greater insight into what their mobile applications are doing, potentially providing guidance as to inappropriate applications to uninstall.

We focused intently on minimizing the overhead of TaintDroid with the goal that performance should not be a barrier to deployment for typical mobile phone users. Unlike existing solutions that rely on heavyweight whole-system emulation [7, 53], we leveraged

the mobile platform's virtualized architecture to integrate three different granularities of taint propagation: variable-level, message-level, and method-level. Though the individual techniques themselves are not new, our contribution lies in the seamless integration of these techniques that provide a winning trade-off between performance and accuracy for constrained smartphone environments. Furthermore, we have integrated multiple taint sources into the information-flow tracking system to automatically label commonly-used sensitive information (e.g., location, microphone, camera, phone numbers). Experiments with our prototype for the Android platform show that the tracking system incurs a runtime overhead of less than 14% for a CPU-bound microbenchmark. More importantly, third-party applications' handling of sensitive data can be monitored with negligible perceived latency.

In addition, we evaluated the accuracy of TaintDroid using 30 randomly selected, popular Android applications that consume location, camera, or microphone data. Without any pre-training or special cases, TaintDroid correctly flagged 105 instances in which these applications transmitted tainted data, and did so without false positives. TaintDroid revealed that 15 of these 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 study showed suspicious use of sensitive data. Our findings demonstrate that TaintDroid can provide a window into the behavior of third-party applications and has the potential to help users discover misbehavior.

Like most other similar information-flow tracking systems [7, 53], TaintDroid can be circumvented through side channels (e.g., leaks via implicit flows [34, 35]). However, the behaviors that create side channels may themselves be atypical behaviors for most applications and may well be detectable through other tools and automated code analysis as we discuss in Section 8. Moreover, the use of side channels to avoid taint detection is, in and of itself, an indicator of overtly malicious intent.

The rest of this paper 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 our TaintDroid design, 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. The smartphone environment presents unique challenges, most important of which is performance. We seize every opportunity to exploit platform properties to build a highly-efficient monitoring system usable for real-time analysis.

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

- *Smartphones are resource constrained devices.* The resource limitations of smartphones precludes the use of heavyweight information tracking systems such as Panorama [53].
- *Third-party applications are partially trusted to send specific types of privacy sensitive information to some but not all network servers.* The monitoring system must distinguish multiple information types, which requires additional computation and storage.
- *Context-based privacy sensitive information is difficult to identify even when sent in the clear.* For example, the user's geographic location is a pair of floating point numbers that frequently changes and is unpredictable. Even if all encryption keys are available, scanning network buffers is ineffective.
- *Applications can share information.* Various forms of information sharing exist, e.g., via files and IPC. Hence, analyzing a single application is insufficient, and a system-wide perspective is required.

We use dynamic taint analysis [53, 41, 8, 56, 36] (also called "taint tracking") to monitor privacy sensitive information on smartphones. When using dynamic taint analysis for privacy, sensitive information is first identified at a *taint source*, which applies a *taint marking* indicating the information type. The analysis tracks how this data impacts other data in a way that might leak the value of the original sensitive information. This tracking is often performed at the instruction level (e.g., add, subtract, etc). Finally, the impacted data is caught before it leaves the system at a *taint sink* (usually the network interface).

Existing taint tracking approaches have several limitations on smartphones. First and foremost, approaches that rely on *instruction-level dynamic taint analysis using whole system emulation [53, 7, 25] incur high performance penalties*. Typically, instruction-level instrumentation incurs 2-20 times slowdown [53, 7] in addition to the slowdown introduced by the emulator, which is not suitable for real-time analysis. Second, *developing accurate taint propagation logic has proven challenging* for the x86 instruction set [37, 45]. Implementa-

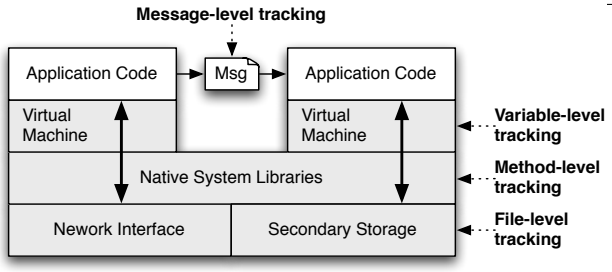


Figure 1: Multi-level approach for performance efficient taint tracking within a common smartphone architecture.

tions of instruction-level tracking experience taint explosion (e.g., if the stack pointer gets falsely tainted) [46] and taint loss (e.g., if complicated instructions such as CMPXCHG, REP MOV are not instrumented properly) [56]. While most smartphones use the ARM instruction set, similar false positives and false negatives will likely arise. Third, *taint tracking implementations commonly only track one taint marking*. Smartphones have many types of privacy sensitive information that must be tracked separately to distinguish legitimate and illegitimate exposure. Tracking multiple taint markings commonly explodes memory consumption, which is beyond reasonable expectations for smartphones.

The taint tracking approach also has advantages for monitoring privacy sensitive information on smartphones. Specifically, many sources of privacy sensitive information on smartphones have well-defined interfaces. For example, all information retrieved from GPS hardware is location information, and all information retrieved from and address book database file is contact information. Commonly, taint tracking systems require heuristics [10] or manual specification [56]. We expand on information sources in Section 5.

Figure 1 presents our approach to taint tracking on smartphones. We leverage architectural features of virtual machine-based smartphones (e.g., Android, BlackBerry, and J2ME-based phones) to enable efficient, system-wide, multiple marking taint tracking. First, we instrument the VM interpreter to provide *variable-level tracking* within untrusted application code.[1] Using variable semantics provided by the interpreter provides valuable context to, for example, avoid the taint explosion observed in the x86 instruction set. Additionally, focusing tracking on variables ensures that 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.

Our approach relies on the firmware's integrity for proper operation. The taint tracking system's trusted computing base includes the virtual machine executing in userspace and any native system libraries loaded by the untrusted interpreted 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.

In summary, we provide a novel efficient, system-wide, multiple-marking, taint tracking design by combining multiple granularities of information tracking. 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 accuracy. As we show in Sections 6 and 7, our system-wide approach is both highly efficient (∼14% overhead) and accurately detects many suspicious network packets.

## 3 Background: Android

Android [1] is a Linux-based, open source, mobile phone platform. Most core phone functionality is implemented as applications running on top of a customized middleware. The middleware itself is written in Java and C/C++. Applications are written in Java and compiled to a custom byte-code known as the Dalvik EXecutable (DEX) byte-code format. Each application executes within its Dalvik VM interpreter instance. Each instance executes as unique UNIX user identities to isolate applications within the Linux platform subsystem. Applications communicate via the binder IPC mechanism. Binder provides transparent message passing based on parcels. We now discuss topics necessary to understand our tracking system.

**Dalvik VM Interpreter:** DEX is a register-based machine language, as opposed to Java byte-code, 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

[1] A similar approach can be applied to just-in-time compilation by inserting tracking code within the generated binary.

types and object references. All computation occurs on registers, therefore values must be loaded from and stored to class fields before use and after use. Note that DEX uses class fields for all long term storage, unlike hardware register-based machine languages (e.g., x86), which store values in arbitrary memory locations.

**Native Methods:** The Android middleware provides access to native libraries for performance optimization and third-party libraries such as OpenGL and Webkit. Android also uses Apache Harmony Java [3], which frequently uses system libraries (e.g., math routines). Native methods are written in C/C++ and expose functionality provided by the underlying Linux kernel and services. They can also access Java internals, and hence are included in our trusted computing base (see 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 [31], 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.

**Binder IPC:** All Android IPC occurs through binder. Binder is a component-based processing and IPC framework designed for BeOS, extended by Palm Inc., and customized for Android by Google. 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 parcel messages 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 data flow 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

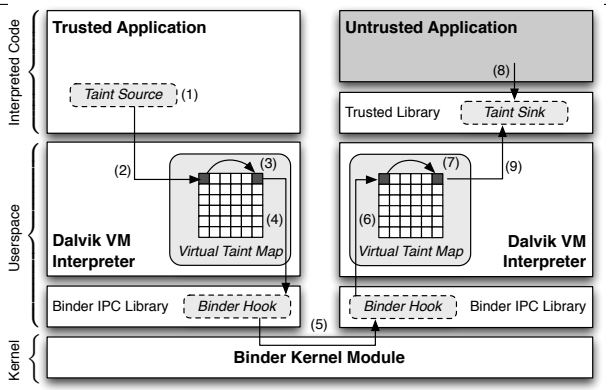


Figure 2: TaintDroid architecture within Android.

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), e.g., 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.

## 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 [53, 7]. Tracked memory is unstructured and without content semantics. Frequently taint tags are stored in non-adjacent shadow memory [53] and tag maps [56]. 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.

Dalvik stores method local variables and arguments on an internal stack. When an application invokes a method, a new stack frame is allocated for all local variables. Method arguments are also passed via the internal

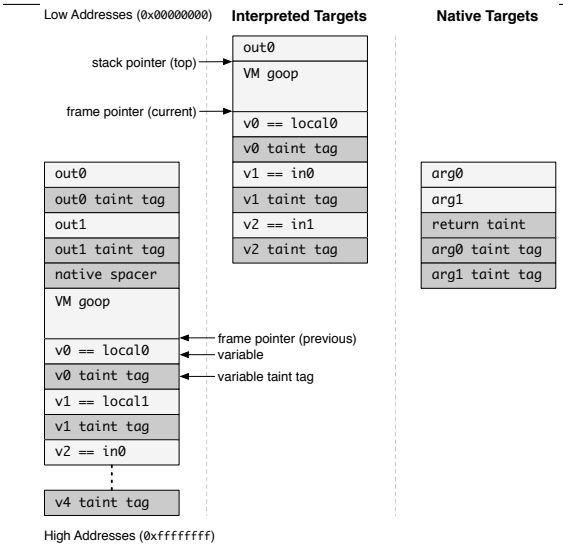


Figure 3: Modified Stack Format. Taint tags are interleaved between registers for interpreted method targets and appended for native methods. Dark grayed boxes represent taint tags.

stack. Before calling a method, the callee places the arguments on the top of the stack such that they become high numbered registers in the callee's stack frame. We allocate taint tag storage by doubling the size of the stack frame allocation. Taint tags are interleaved between values such that register $v_i$ originally accessed via $fp[i]$ is accessed as $fp[2 \cdot i]$ after modification. Note that Dalvik stores 64-bit variables as two adjacent 32-bit registers. We do not differentiate between 32-bit and 64-bit variables to simplify stack frame access. Furthermore, native method targets require a slightly different stack frame organization for reasons discussed in Section 4.3. The modified stack format is shown in Figure 3.

Taint tags are stored adjacent to class fields and arrays inside the VM interpreter's internal data structures. 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 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 to $A[0]$, $v$ will be tainted, even though $u$ was untainted. Fortunately, Java frequently uses objects, and object references are infrequently tainted (see Section 4.2), therefore such false positives are intuitively minimized.

## 4.2 Interpreted Code Taint Propagation

Taint tracking granularity and flow semantics influence performance and accuracy. TaintDroid implements variable-level taint tracking within the Dalvik VM interpreter. Variables provide valuable semantics for taint propagation, distinguishing data pointers from integer values. TaintDroid primarily tracks primitive type variables (e.g., *int*, *float*, etc); however, there are cases when object references must become tainted to ensure taint propagation operates correctly.

The Dalvik VM operates on the unique DEX machine language instruction set, therefore we must design an appropriate propagation logic. We use a data flow logic, as tracking implicit flows requires static analysis and causes significant performance overhead and overestimation in tracking [28] (see Section 8). We begin by defining taint markings, taint tags, variables, and taint propagation. We then present our logic rules for DEX.

Let $\mathcal{L}$ be the universe of taint markings for a particular system. A taint tag $t$ is a set of taint markings, $t \in \mathcal{L}$. Each variable has an associated taint tag. A variable is an instance of one of the five types described in Section 4.1. We use a different representation for each type. 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.

Our virtual taint map function is $\tau(\cdot)$. $\tau(v)$ returns the taint tag $t$ for variable $v$. $\tau(v)$ is also used to assign a taint tag to a variable. Retrieval and assignment is distinguished by the position of $\tau(\cdot)$ w.r.t. 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 $v_2$ to $v_1$.

Table 1 captures our propagation logic. The table enumerates abstracted versions of the byte-code 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 maintained within the interpreter, respectively. $A$, $B$, and $C$ are constants in the byte-code. The table does not list instructions that clear the taint tag of the destination register. For example, we do not consider the *array-length* instruction to return a tainted value even if the array is tainted. Note that the array length is sometimes used to aid direct control flow propagation (e.g., Vogt et al. [50]).

The propagation rules are straightforward with one exception. Taint propagation logics commonly include the taint tag of an array index during lookup to handle translation tables. However, 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

Table 1: DEX Taint Propagation Logic. 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 byte-code constants.

| 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 field $f_C$ and object reference taint |

value. Therefore, we include the object reference taint tag in the instance *get* (*iget-op*) rule. This technique successfully propagates the taint tag from the array index to the value of the object (e.g., the *Integer* value).

## 4.3 Native Code Taint Propagation

Native code is unmonitored in TaintDroid. Ideally, we achieve the same propagation semantics as the interpreted counterpart. Hence, we define two *necessary postconditions* for accurate taint tracking in the Java-like environment: 1) all accessed external variables (i.e., class fields referenced by other methods) are assigned taint tags according to data flow rules; and 2) the return value is assigned a taint tag according to data flow rules. TaintDroid achieves these postconditions through an assortment of manual instrumentation, heuristics, and method profiles, depending on situational requirements.

**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. We manually inspected and patched Dalvik's internal VM methods for taint propagation as needed. We identified 185 internal VM methods in Android version 2.1; however, only 5 required patching: the *System.arraycopy()* native method for copying array contents, and several native methods implementing Java reflection. Correctness was verified experimentally.

**JNI Methods:** JNI methods are invoked through the JNI call bridge. The call bridge parses Java arguments and assigns a return value using the method's descriptor string. We patched the call bridge to provide taint propagation for all JNI methods. When a JNI method returns, TaintDroid consults a method profile table for tag propagation updates. A method profile is a list of $(from, to)$ pairs indicating flows between variables, which may be method parameters, class variables, or return values. Enumerating the information flows for all JNI methods is a time consuming task best completed automatically using source code analysis (a task we leave for future work). We currently include an additional propagation heuristic patch. The heuristic is conservative for JNI methods that only operate on primitive and String arguments and return values. It assigns the union of the method argument taint tags and to the taint tag of the return value. While the heuristic has false negatives for methods using objects, it covers many existing methods.

We performed a survey of the JNI methods included in the official Android source code (version 2.1) to determine specific properties. We found 2,844 JNI methods with a Java interface and C or C++ implementation.[2] Of these methods, 913 did not reference objects (as arguments, return value, or method body) and hence are automatically covered by our heuristic. The remaining methods may or may not have information flows that produce false negatives. Currently, we define method profiles as needed. For example, methods in the IBM *NativeConverter* class require propagation for conversion between character and byte arrays.

## 4.4 IPC Taint Propagation

Taint tags must propagate between applications when they exchange data. The tracking granularity affects performance and memory overhead. TaintDroid uses message-level taint tracking. A message taint tag represents the upper bound of taint markings assigned to vari-

[2]There was a relatively small number of JNI methods that did not either have a Java interface or C/C++ implementation. These unusable methods were excluded from our survey.

ables contained in the message. We use message-level granularity to minimize performance and storage overhead during IPC.

Message-level taint propagation for IPC can lead to false positives. Similar to arrays, all data items in a parcel share the same taint tag. At the expense of additional memory and performance overhead, a shadow parcel containing taint tags for each 32-bit value would remove these false positives.

### 4.5 Secondary Storage Taint Propagation

Taint tags may be lost when data is written to a file. Our design stores one taint tag per file. The taint tag is updated on file write and propagated to data on file read. TaintDroid stores file taint tags in the file system's extended attributes. To do this, we implemented extended attribute support for Android's host file system (YAFFS2) and formatted the removable SDcard with the ext2 file system. As with arrays and IPC, storing one taint tag per file leads to false positives. 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. The interface performs two functions: 1) add taint markings to variables; and 2) retrieve 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. Primitive type variables, on the other hand, are stored on the interpreter's internal stack and disappear after a method is called. Therefore, the taint library uses the method return value as a means of tainting primitive type variables. The developer passes a value or variable into the appropriate add taint method (e.g., *addTaintInt()*) and the returned variable has the same value but additionally has the specified taint tag. Note that the stack storage does not pose complications for taint tag retrieval.

## 5 Privacy Hook Placement

Using TaintDroid for privacy analysis requires identifying privacy sensitive sources and instrumenting taint sources within the operating system. Historically, dynamic taint analysis systems assume taint source and sink placement is trivial. However, complex operating systems such as Android provide applications information in a variety of ways, e.g., direct access, and service interface. Each potential type of privacy sensitive information must be studied carefully to determine the best method of defining the taint source. Our design decision to track information within the VM interpreter also limits placement of potential hooks: native code often cannot communicate tags to interpreted code. We now discuss four general taint source types and our taint sink.

**Low-bandwidth Sensors:** A variety of privacy sensitive information types are acquired through low-bandwidth sensors, e.g., 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 applications.

**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. We used placed hooks for both data buffer and file tainting to track microphone and camera information.

**Information Databases:** Shared information such as address books and SMS messages are often stored in file-based databases. This organization provides a useful unambiguous taint source similar to hardware sensors. By adding a taint tag to such database files, all information read from the file will be automatically tainted. We used this technique for tracking address book information.

**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 transmits out the network interface. The VM interpreter-based approach requires the taint sink to 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 an application study that uses TaintDroid to analyze how third-party Android applications use privacy sensitive user data. Existing applications make use of a variety of user data along with permissions to access the Internet. Our study finds that two thirds of these applications expose detailed location data, the phone's unique ID, and the phone number using the combination of the seemingly innocuous access permissions granted at install. This finding was made possible by TaintDroid's ability to monitor runtime *access* of sensitive user data and to precisely relate the monitored accesses with the *data exposure* by applications.

### 6.1 Experimental Setup

An early 2010 survey of the 50 most popular free applications in each category of the Android Market [2] (1100 applications, in total) revealed that roughly a third of the applications (32.8%) require Internet permissions along with permissions to access either location, camera, or audio data. From this set, we randomly selected 30 popular applications spanning twelve categories: Table 2 enumerates these applications along with permissions they request at install time. Note that this does not reflect actual access or use of sensitive data.

We studied each of the thirty downloaded applications by starting the application, performing any initialization or registration that was required, and then manually exercising the functionality offered by the application. We recorded system logs including detailed information from TaintDroid: tainted binder messages, tainted file output, and tainted network messages with the remote address. The overall experiment (conducted in May 2010) lasted slightly over 100 minutes, generating 22,594 packets (8.6MB) and 1,130 TCP connections. 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 the same version of TaintDroid built on Android 2.1. Though the phones used for experiments had a valid SIM card installed, the SIM card was inactivated, 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 information to identify 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, but disclosing the coordinates to an advertisement server is a privacy violation unless explicitly accepted in a terms of use agreement.

### 6.2 Findings

Table 3 summarizes our findings. TaintDroid flagged 105 TCP connections as containing tainted privacy sensitive information. We manually labeled each message based on available context, including remote server names and temporally relevant application log messages. We used remote hostnames as an indication of whether data was being sent to a server providing application functionality or to a third party. Frequently, messages contained plaintext that aided categorization, e.g., an HTTP GET request containing geographic coordinates. However, 21 flagged messages contained binary data. Our investigation indicates these messages were generated by the Google Maps for Mobile [20] and FlurryAgent [19] APIs and contained tainted privacy sensitive data. These conclusions are supported by message transmissions immediately after the application received a tainted parcel from the system location manager. We now expand on our findings for each category and reflect on potential privacy violations.

**Phone Information:** Table 2 shows that 21 out of the 30 applications require permissions to read phone state and the Internet. We found that 2 of the 21 applications transmitted to their server (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 which is a unique SIM card serial number. We verified messages were flagged correctly by inspecting the plaintext payload.[3]

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 transmits the phone information *every time* the phone boots. While this application displays a terms of use on first use, the terms of use does not specify collection of this highly sensitive data. Surprisingly, this application transmits the phone data immediately after install, before first use.

**Device Unique ID:** The device's IMEI was also exposed by applications. The IMEI uniquely identifies a specific mobile phone and is used to prevent a stolen handset from accessing the cellular network. TaintDroid flags indicated that nine applications transmitted the IMEI. Seven out of the nine applications either do not present an End User License Agreement (EULA) or do not specify IMEI collection in the EULA. One of the seven applications is a popular social networking application and

[3] 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.

Table 2: Applications grouped by the requested permissions (L: location, C: camera, A: audio, P: phone state). Android Market categories are indicated in parenthesis, showing the diversity of the studied applications.

| Applications | # | Permissions* | | | |
|---|---|---|---|---|---|
| | | L | C | A | P |
| The Weather Channel (News & Weather); Cestos, Solitaire (Game); Movies (Entertainment); Babble (Social); Manga Browser (Comics) | 6 | x | | | |
| Bump, Wertago (Social); Antivirus (Communication); ABC — Animals, Traffic Jam, Hearts, Blackjack, (Games); Horoscope (Lifestyle); 3001 Wisdom Quotes Lite, Yellow Pages (Reference); Dastelefonbuch, Astrid (Productivity), BBC News Live Stream (News & Weather); Ringtones (Entertainment) | 14 | x | | | x |
| Layer (Productivity); Knocking (Social); Barcode Scanner, Coupons (Shopping); Trapster (Travel); Spongebob Slide (Game); ProBasketBall (Sports) | 7 | x | x | | x |
| MySpace (Social); ixMAT (Shopping) | 2 | | x | | |
| Evernote (Productivity) | 1 | x | x | x | |

* All listed applications also require access to the Internet.

Table 3: Potential privacy violations by 20 of the studied applications. Note that three applications had multiple violations, one of which had a violation in all three categories.

| Observed Behavior (# of apps) | Details |
|---|---|
| Phone Information to Content Servers (2) | 2 apps sent out the phone number, IMSI, and ICC-ID along with the geo-coordinates to the app's content server. |
| Device ID to Content Servers (7) | 2 Social, 1 Shopping, 1 Reference and three other apps transmitted the IMEI number to the app's content server. |
| Location to Advertisement Servers (15) | 5 apps sent geo-coordinates 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. |

*To the best of our knowledge, the binary messages contained tainted location data. See the discussion below.

another is a location-based search application. Furthermore, we found two of the seven applications include the IMEI when transmitting the device's geographic coordinates to their content server, potentially repurposing the IMEI as a client ID.

In comparison, two of the nine applications treat the IMEI with proper care, thus we do not classify them as potential privacy violators. One application displays a privacy statement that clearly indicates that the application collects the device ID. The other uses the hash of the IMEI instead of the number itself. We verified this practice by comparing results from two different phones.

**Location Data to Advertisement Servers:** Half of the studied applications exposed location data to third-party advertisement servers without requiring implicit or explicit user consent. Of the fifteen applications, only two presented a EULA on first run; however neither EULA indicated this practice. Without explicit or implicit consent, these flags reflect potential privacy violations. 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 (4 applications). The plaintext location exposure to AdMob occurred in the HTTP GET string:

```
...&s=a14a4a93f1e4c68&..&t=062A1CB1D476DE85
B717D9195A6722A9&d%5Bcoord%5D=47.6612278900
00006%2C-122.31589477&...
```

Investigating the AdMob SDK revealed the `s=` parameter is an identifier unique to an application 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.

Our experimentation indicates these fifteen applications collect location data for the sole purpose of sending it to advertisement servers. In some cases, location data was transmitted to advertisement servers even when no advertisement was displayed in the application. However, we note that TaintDroid helped us verify that three of the studied applications (not included in the Table 3) only transmitted location data per user's request to pull localized content from their servers. This finding demon-

strates the importance of monitoring exercised functionality of an application that reflects how the application *actually* uses or abuses the granted permissions.

**Legitimate Flags:** Out of 105 connections flagged by TaintDroid, 37 were deemed legitimate use. The flags resulted from four applications and the OS itself while using the Google Maps for Mobile (GMM) API. The TaintDroid 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, which is difficult to verify with the lack of the source code of the third-party applications.

**Summary:** Our study of 30 popular applications 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 by the tested applications. Half of the studied applications share location data with advertisement servers. Approximately one third of the applications expose the device ID, sometimes with the phone number and the SIM card serial number. The analysis was simplified by the taint tag provided by TaintDroid that precisely describes which privacy relevant data is included in the payload, especially for binary payloads. We also note that there was almost no perceived latency while running experiments with TaintDroid.

## 7 Performance Evaluation

We now study TaintDroid's taint tracking overhead. Experiments were performed on a Google Nexus One running Android OS version 2.1 modified for TaintDroid. 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 file access includes appropriate taint tags.

### 7.1 Macrobenchmarks

For all but a few tested applications, we were anecdotally unable to perceive significant 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 webpage rendering) occur in unmonitored native libraries.

To gain further insight into perceived overhead, we devised five macrobenchmarks for common 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 4.

Table 4: Macrobenchmark Results

| | Android | TaintDroid |
|---|---|---|
| **App Load Time** | 63 ms | 65 ms |
| **Address Book (create)** | 348 ms | 367 ms |
| **Address Book (read)** | 101 ms | 119 ms |
| **Phone Call** | 96 ms | 106 ms |
| **Take Picture** | 1718 ms | 2216 ms |

**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 3% 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 SQL transactions. The subsequent delete operation was lazy, returning in 0 ms, and hence was excluded from our results. TaintDroid adds approximately 5.5% and 18% overhead for address book entry creates and reads, respectively. The additional overhead for reads can be attributed to file taint propagation. The data is not tainted before create, hence no file propagation is needed. Note that the user experiences less than 20 ms 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 only adds 10 ms per phone call setup ($\sim$10% overhead), which is significantly less than call setup in the network, which 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 re-enabled. This measurement includes the time to capture a picture from the camera and save the file to the SDcard. TaintDroid observes approximately 29% overhead when taking a picture. Note that the file write requires file taint propagation for *each* data buffer. While 498 ms overhead per picture is noticeable, it is acceptable for smartphone picture takers who do not capture images in rapid succession. Note that this overhead can be reduced by eliminating redundant propagation.

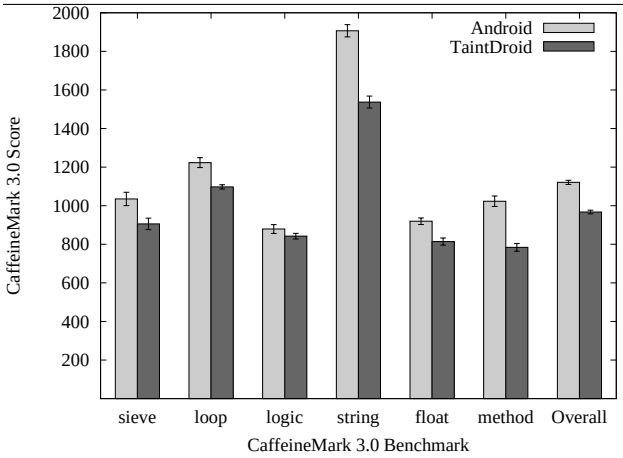


Figure 4: Microbenchmark of Java overhead. Error bars indicate 95% confidence intervals.

## 7.2 Java Microbenchmark

Figure 4 shows the execution time results of a Java microbenchmark. We used an Android port of the standard CaffeineMark 3.0 [40]. CaffeineMark uses an internal scoring metric only useful for relative comparisons.

The results are consistent with implementation-specific expectations. The overhead incurred by TaintDroid is smallest for the benchmarks dominated by arithmetic and logic operations. The taint propagation for these operations is simple, consisting of an additional copy of spatially local memory. The string benchmark, on the other hand, experiences the greatest overhead. This is most likely due to the JNI propagation heuristic overhead when arguments reference String objects.

The "overall" results indicate cumulative score across individual benchmarks. CaffeineMark documentation states that scores roughly correspond to the number of Java instructions executed per second. Here, the unmodified Android system had an average score of 1121, and TaintDroid measured 967. TaintDroid has a 14% overhead with respect to the unmodified system.

We also measured memory consumption during the CaffeineMark benchmark. The benchmark consumed 21.28 MB on the unmodified system and 22.21 MB while running on TaintDroid, indicating a 4.4% memory overhead. Given that TaintDroid stores 32 taint markings (4 bytes) for each 32-bit variable (regardless of taint state), this overhead is expected.

## 7.3 IPC Microbenchmark

The IPC benchmark considers overhead due to the parcel modifications. For this experiment, we developed client and service applications that perform binder transactions as fast as possible. The service manipulates account objects (a username string and a balance integer) and provides two interfaces: *setAccount()* and *getAccount()*. The experiment measures the time for the client to invoke each interface pair 10,000 times.

Table 5: IPC Benchmark Results.

| | **Android** | **TaintDroid** |
|---|---|---|
| Time (s) | 8.58 | 10.89 |
| Memory (client) | 21.06MB | 21.88MB |
| Memory (service) | 18.92MB | 19.48MB |

Table 5 summarizes the results of the IPC benchmark. TaintDroid was 27% slower than Android. TaintDroid only adds four bytes to each IPC object, therefore overhead due to data size is unlikely. The 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. Finally, TaintDroid used 3.5% more memory than Android, which is comparable to the consumption observed during the CaffeineMark benchmarks.

# 8 Discussion

**Approach Limitations:** TaintDroid only tracks data flows (i.e., explicit flows) and does not track control flows (i.e., implicit flows) to minimize performance overhead. Section 6 shows that TaintDroid can track applications' expected data exposure and also reveal suspicious actions. However, applications that are truly malicious can game our system and exfiltrate privacy sensitive information through control flows. Fully tracking control flow requires static analysis [14, 34], which is not applicable to analyzing third-party applications whose source code is unavailable. Direct control flows can be tracked dynamically if a taint scope can be determined [50]; however, DEX does not maintain branch structures that TaintDroid can leverage. On-demand static analysis to determine method control flow graphs (CFGs) provides this context [36]; however, TaintDroid does not currently perform such analysis in order to avoid false positives and significant performance overhead. Our data flow taint propagation logic is consistent with existing, well known, taint tracking systems [7, 53]. Finally, once information leaves the phone, it may return in a network reply. TaintDroid cannot track such information.

**Implementation Limitations:** Android uses the Apache Harmony [3] 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 oc-

curred 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 observes 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), which are all tainted together.[4] 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.

## 9 Related Work

Mobile phone host security is a growing concern. OS-level protections such as Kirin [17], Saint [39], and Security-by-Contract [15] provide enhanced security mechanisms for Android and Windows Mobile. These approaches are designed to prevent access to sensitive information; however, once information enters the application, no additional mediation occurs. In systems with larger displays, a graphical widget [26] can help users visualize sensor access policies. Mulliner et al. [33] provide information tracking by labeling smartphone processes based on the interfaces they access. Policy enforcement prohibits processes from accessing subsequent interfaces based on label assignment.

Decentralized information flow control (DIFC) enhanced operating systems such as Asbestos [49] and HiStar [55] label processes and enforce access control based on Denning's lattice model for information flow security [13]. Flume [29] provides similar enhancements for legacy OS abstractions. Related, PRECIP [51] 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 [27] and TightLip [54]. 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 [43] extends existing programming languages by labeling variables with security attributes. Compilers use the security labels to generate security proofs, e.g., Jif [34, 35] and SLam [23]. Laminar [42] provides DIFC guarantees based on programmer defined security regions. However, these languages require careful development and are often incompatible with legacy software designs [24].

Dynamic taint analysis provides information tracking for legacy programs. The approach has been used to enhance system integrity (e.g., defend against software attacks [38, 41, 8]) and confidentiality (e.g., discover privacy exposure [53, 16, 56]), as well as track Internet worms [9]. Dynamic tracking approaches range from whole-system analysis using hardware extensions [48, 11, 47] and emulation environments [7, 53] to per-process tracking using dynamic binary translation (DBT) [6, 41, 8, 56]. The performance and memory overhead associated with dynamic tracking has resulted in an array of optimizations, including optimizing context switches [41], on-demand tracking [25] based on hypervisor introspection, and function summaries for code with known information flow properties [56]. If source code is available, significant performance improvements can be achieved by automatically instrumenting legacy programs with dynamic tracking functionality [52, 30]. Automatic instrumentation has also been performed on x86 binaries [44], providing a compromise between source code translation and DBT. Our TaintDroid design was inspired by these prior works, but addressed different challenges unique to mobile phones. Moreover, we leverage architectural features to avoid instruction-level taint tracking, which incurs high performance overhead.

Finally, dynamic taint analysis has been applied to virtual machines and interpreters. Haldar et al. [21] instrument the Java String class with taint tracking to prevent SQL injection attacks. WASP [22] has similar motivations; however, it uses positive tainting of individual characters to ensure the SQL query contains only high-integrity substrings. Chandra and Franz [5] propose fine-grained information flow tracking within the JVM and instrument Java byte-code to aid control flow analysis. Similarly, Nair et al. [36] instrument the Kaffe JVM. Vogt et al. [50] instrument a Javascript interpreter to prevent cross-site scripting attacks. Finally, Xu et al. [52] automatically instrument the PHP interpreter source code with dynamic information tracking to prevent SQL injection attacks. TaintDroid's interpreted code taint propagation bears similarity to some of these works. However, TaintDroid is the first system that implements system-wide information flow tracking, seamlessly connecting

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

interpreter taint tracking with the rest of the platform.

## 10 Conclusions

While some mobile phone operating systems allow users to control applications' access to sensitive information, such as location sensors, camera images, and contact lists, users lack visibility into how applications use their private data. To address this, we present TaintDroid, an efficient, system-wide information flow tracking tool that can simultaneously track multiple sources of sensitive data. A key design goal of TaintDroid is efficiency, and TaintDroid achieves this by integrating four granularities of taint propagation (variable-level, message-level, method-level, and file-level) to achieve a 14% performance overhead on a CPU-bound microbenchmark.

We also used our TaintDroid implementation to study the behavior of 30 popular third-party applications, chosen at random from the Android Marketplace. Our study revealed that 15 of the 30 applications reported users' locations to remote advertising servers, and that two-thirds of the applications in our study exhibit suspicious handling of sensitive data. Our findings demonstrate the effectiveness and value of enhancing smartphone platforms with monitoring tools such as TaintDroid.

## References


[1] Android. http://www.android.com.

[2] Android Market. http://market.android.com.

[3] Apache Harmony – Open Source Java Platform. http://harmony.apache.org.

[4] Apple, Inc. Apple's App Store Downloads Top Two Billion. http://www.apple.com/pr/library/2009/09/28appstore.html, September 28, 2009.

[5] D. Chandra and M. Franz. Fine-Grained Information Flow Analysis and Enforcement in a Java Virtual Machine. In *Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC)*, December 2007.

[6] W. Cheng, Q. Zhao, B. Yu, and S. Hiroshige. TaintTrace: Efficient Flow Tracing with Dyanmic Binary Rewriting. In *Proceedings of the IEEE Symposium on Computers and Communications (ISCC)*, pages 749–754, June 2006.

[7] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum. Understanding Data Lifetime via Whole System Simulation. In *Proceedings of the 13th USENIX Security Symposium*, August 2004.

[8] J. Clause, W. Li, and A. Orso. Dytan: A Generic Dynamic Taint Analysis Framework. In *Proceedings of the 2007 international symposium on Software testing and analysis*, pages 196–206, 2007.

[9] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: End-to-End Containment of Internet Worms. In *Proceedings of the ACM Symposium on Operating Systems Principles*, 2005.

[10] L. P. Cox and P. Gilbert. RedFlag: Reducing Inadvertent Leaks by Personal Machines. Technical Report TR-2009-02, Duke University, 2009.

[11] J. R. Crandall and F. T. Chong. Minos: Control Data Attack Prevention Orthogonal to Memory Model. In *Proceedings of the International Symposium on Microarchitecture*, pages 221–232, December 2004.

[12] C. Davies. iPhone spyware debated as app library "phones home". http://www.slashgear.com/iphone-spyware-debated-as-app-library-phones-home-1752491/, August 17, 2009.

[13] D. E. Denning. A Lattice Model of Secure Information Flow. *Communications of the ACM*, 19(5):236–243, May 1976.

[14] D. E. Denning and P. J. Denning. Certification of Programs for Secure Information Flow. *Communications of the ACM*, 20(7), July 1977.

[15] L. Desmet, W. Joosen, F. Massacci, P. Philippaerts, F. Piessens, I. Siahaan, and D. Vanoverberghe. Security-by-contract on the .NET platform. *Information Security Technical Report*, 13(1):25–32, January 2008.

[16] M. Egele, C. Kruegel, E. Kirda, H. Yin, and D. Song. Dyanmic Spyware Analysis. In *Proceedings of the USENIX Annual Technical Conference*, pages 233–246, June 2007.

[17] W. Enck, M. Ongtang, and P. McDaniel. On Lightweight Mobile Phone Application Certification. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS)*, November 2009.

[18] M. Fitzpatrick. Mobile that allows bosses to snoop on staff developed. BBC News, March 2010. http://news.bbc.co.uk/2/hi/technology/8559683.stm.

[19] Flurry Mobile Application Analytics. http://www.flurry.com/product/technical-info.html.

[20] Google Maps for Mobile. http://www.google.com/mobile/products/maps.html.

[21] V. Haldar, D. Chandra, and M. Franz. Dynamic Taint Propagation for Java. In *Proceedings of the 21st Annual Computer Security Applications Conference (ACSAC)*, pages 303–311, December 2005.

[22] W. G. Halfond, A. Orso, and P. Manolios. WASP: Protecting Web Applications Using Positive Tainting and Syntax-Aware Evaluation. *IEEE Transactions on Software Engineering*, 34(1):65–81, 2008.

[23] N. Heintze and J. G. Riecke. The SLam Calculus: Programming with Secrecy and Integrity. In *Proceedings of the Symposium on Principles of Programming Languages (POPL)*, pages 365–377, 1998.

[24] B. Hicks, K. Ahmadizadeh, and P. McDaniel. Understanding practical application development in security-typed languages. In *22st Annual Computer Security Applications Conference (ACSAC)*, pages 153–164, 2006.

[25] A. Ho, M. Fetterman, C. Clark, A. Warfield, and S. Hand. Practical Taint-Based Protection using Demand Emulation. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, pages 29–41, 2006.

[26] J. Howell and S. Schechter. What You See is What they Get: Protecting users from unwanted use of microphones, camera, and other sensors. In *Proceedings of Web 2.0 Security and Privacy Workshop*, 2010.

[27] J. Jung, A. Sheth, B. Greenstein, D. Wetherall, G. Maganis, and T. Kohno. Privacy Oracle: A System for Finding Application Leaks with Black Box Differential Testing. In *Proceedings of ACM CCS*, 2008.

[28] D. King, B. Hicks, M. Hicks, and T. Jaeger. Implicit Flows: Can't Live with 'Em, Can't Live without 'Em. In *Proceedings of the International Conference on Information Systems Security*, 2008.

[29] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information Flow Control for Standard OS Abstractions. In *Proceedings of ACM Symposium on Operating Systems Principles*, 2007.

[30] L. C. Lam and T. cker Chiueh. A General Dynamic Information Flow Tracking Framework for Security Applications. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2006.

[31] S. Liang. *Java Native Interface: Programmer's Guide and Specification*. Prentice Hall PTR, 1999.

[32] D. Moren. Retrievable iPhone numbers mean potential privacy issues. `http://www.macworld.com/article/143047/2009/09/phone_hole.html`, September 29, 2009.

[33] C. Mulliner, G. Vigna, D. Dagon, and W. Lee. Using Labeling to Prevent Cross-Service Attacks Against Smart Phones. In *Proceedings of Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2006.

[34] A. C. Myers. JFlow: Practical Mostly-Static Information Flow Control. In *Proceedings of the ACM Symposium on Principles of Programming Langauges (POPL)*, January 1999.

[35] A. C. Myers and B. Liskov. Protecting Privacy Using the Decentralized Label Model. *ACM Transactions on Software Engineering and Methodology*, 9(4):410–442, October 2000.

[36] S. K. Nair, P. N. Simpson, B. Crispo, and A. S. Tanenbaum. A Virtual Machine Based Information Flow Control System for Policy Enforcement. In *the 1st International Workshop on Run Time Enforcement for Mobile and Distributed Systems (REM)*, 2007.

[37] J. Newsome, S. McCamant, and D. Song. Measuring channel capacity to distinguish undue influence. In *ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, 2009.

[38] J. Newsome and D. Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Proc. of Network and Distributed System Security Symposium*, 2005.

[39] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel. Semantically Rich Application-Centric Security in Android. In *Proceedings of the 25th Annual Computer Security Applications Conference (ACSAC)*, 2009.

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

[41] F. Qin, C. Wang, Z. Li, H. seop Kim, Y. Zhou, and Y. Wu. 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*, pages 135–148, 2006.

[42] I. Roy, D. E. Porter, M. D. Bond, K. S. McKinley, and E. Witchel. Laminar: Practical Fine-Grained Decentralized Information Flow Control. In *Proceedings of Programming Language Design and Implementation*, 2009.

[43] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communication*, 21(1):5–19, January 2003.

[44] P. Saxena, R. Sekar, and V. Puranik. Efficient Fine-Grained Binary Instrumentation with Applications to Taint-Tracking. In *Proceedings of the IEEE/ACM symposium on Code Generation and Optimization (CGO)*, 2008.

[45] E. J. Schwartz, T. Avgerinos, and D. Brumley. All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but might have been afraid to ask). In *IEEE Symposium on Security and Privacy*, 2010.

[46] A. Slowinska and H. Bos. Pointless Tainting? Evaluating the Practicality of Pointer Tainting. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, pages 61–74, April 2009.

[47] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure Program Execution via Dynamic Information Flow Tracking. In *Proceedings of Architectural Support for Programming Languages and Operating Systems*, 2004.

[48] N. Vachharajani, M. J. Bridges, J. Chang, R. Rangan, G. Ottoni, J. A. Blome, G. A. Reis, M. Vachharajani, and D. I. August. RIFLE: An Architectural Framework for User-Centric Information-Flow Security. In *Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, pages 243–254, 2004.

[49] S. Vandebogart, P. Efstathopoulos, E. Kohler, M. Krohn, C. Frey, D. Ziegler, F. Kaashoek, R. Morris, and D. Mazières. Labels and Event Processes in the Asbestos Operating System. *ACM Transactions on Computer Systems (TOCS)*, 25(4), December 2007.

[50] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross-Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In *Proc. of Network & Distributed System Security*, 2007.

[51] X. Wang, Z. Li, N. Li, and J. Y. Choi. PRECIP: Towards Practical and Retrofittable Confidential Information Protection. In *Proceedings of 15th Network and Distributed System Security Symposium (NDSS)*, 2008.

[52] W. Xu, S. Bhatkar, and R. Sekar. Taint-Enhanced Policy Enforcement: A Practical Approach to Defeat a Wide

Range of Attacks. In *Proceedings of the USENIX Security Symposium*, pages 121–136, August 2006.

[53] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: Capturing System-wide Information Flow for Malware Detection and Analysis. In *Proceedings of ACM Computer and Communications Security*, 2007.

[54] A. R. Yumerefendi, B. Mickle, and L. P. Cox. TightLip: Keeping Applications from Spilling the Beans. In *Proceedings of the 4th USENIX Symposium on Network Systems Design & Implementation (NSDI)*, 2007.

[55] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making Information Flow Explicit in HiStar. In *Proceedings of the 7th symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[56] D. Zhu, J. Jung, D. Song, T. Kohno, and D. Wetherall. Privacy Scope: A Precise Information Flow Tracking System For Finding Application Leaks. Technical Report EECS-2009-145, Department of Computer Science, UC Berkeley, 2009.