Networking and Security Research Center

Technical Report

NAS-TR-0120-2010

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

William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox,
Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth

11 October 2010

The Pennsylvania State University
344 IST Building
University Park, PA 16802 USA
http://nsrc.cse.psu.edu

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 frequently fail to provide users with adequate control over and visibility into how third-party applications use 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 provides realtime analysis by leveraging Android's virtualized execution environment. TaintDroid incurs only 14% performance overhead on a CPU-bound micro-benchmark and imposes negligible overhead on interactive third-party applications. Using TaintDroid to monitor the behavior of 30 popular third-party Android applications, we found 68 instances of potential misuse of users' private information across 20 applications. Monitoring sensitive data with TaintDroid provides informed use of third-party applications for phone users and valuable input for smartphone security service firms seeking to identify misbehaving applications.

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 nearly 3 billion applications after only 18 months [4]. 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 [35, 12] and the privacy risks posed by seemingly innocent sensors like accelerometers [19] 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 paper 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 [33, 55].

Analysis of applications' behavior requires sufficient contextual information about what data leaves a device and where it is sent. Thus, TaintDroid automatically labels (taints) data from privacy-sensitive sources and transitively applies labels as sensitive data propagates through program variables, files, and interprocess messages. When tainted data are transmitted over the network, or otherwise leave 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

*This Technical Report contains the implementation details excluded from the conference version appearing in the proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI'10).

identify misbehaving applications.

To be practical, the performance overhead of the TaintDroid runtime must be minimal. Unlike existing solutions that rely on heavy-weight whole-system emulation [7, 57], 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 accuracy for resource constrained smartphones. Experiments with our prototype for Android show that tracking incurs a runtime overhead of less than 14% for a CPU-bound microbenchmark. More importantly, interactive third-party applications can be monitored with negligible perceived latency.

We evaluated the accuracy of TaintDroid using 30 randomly selected, popular Android applications that use location, camera, or microphone data. TaintDroid correctly flagged 105 instances in which these applications transmitted tainted data; of the 105, we determined that 37 were clearly legitimate. TaintDroid also 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 study used sensitive data suspiciously. Our findings demonstrate that TaintDroid can help expose potential misbehavior by third-party applications.

Like similar information-flow tracking systems [7, 57], 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 [14, 46] as we discuss in Section 8.

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 in realtime. Many smartphone applications are closed-source, therefore, static source code analysis is infeasible. Even if source code is available, runtime events and configuration often dictate information use; realtime monitoring accounts for these environment specific dependencies.

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

- *Smartphones are resource constrained.* The resource limitations of smartphones precludes the use of heavyweight information tracking systems such as Panorama [57].
- *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 [57, 44, 8, 61, 39] (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 impacted data is identified before it leaves the system at a *taint sink* (usually the network interface).

Existing taint tracking approaches have several limitations. First and foremost, approaches that rely on instruction-level dynamic taint analysis using whole system emulation [57, 7, 26] incur high performance penalties. Instruction-level instrumentation incurs 2-20 times slowdown [57, 7] in addition to the slowdown introduced by emulation, which is not suitable for realtime analysis. Second, developing accurate taint propagation logic has proven challenging for the x86 instruction set [40, 48]. Implementations of instruction-level tracking can experience taint explosion if the stack pointer becomes falsely tainted [49] and taint loss if complicated instructions such as CMPXCHG, REP MOV are not instrumented properly [61]. 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, Black-

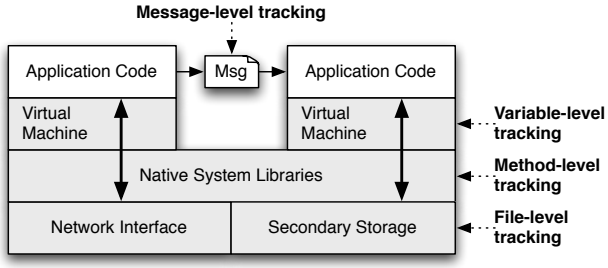

Figure 1: Multi-level approach for performance efficient taint tracking within a common smartphone architecture.

Berry, and J2ME-based phones) 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 untrusted application code.[1] 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 [10] or manual specification [61] 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 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. Note that an early 2010 survey of the top 50 most popular free applications in each category of the Android Market [2] (1100 applications in total) revealed that less than 4% included a `.so` file. A similar survey conducted in mid 2010 revealed this fraction increased to 5%, which indicates there is growth in the number of applications using native third-party libraries, but that the number of affected applications remains small.

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 ($\sim$14% CPU overhead and $\sim$4.4% memory overhead for simultaneously tracking 32 taint markings per data unit) 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 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 fre-

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

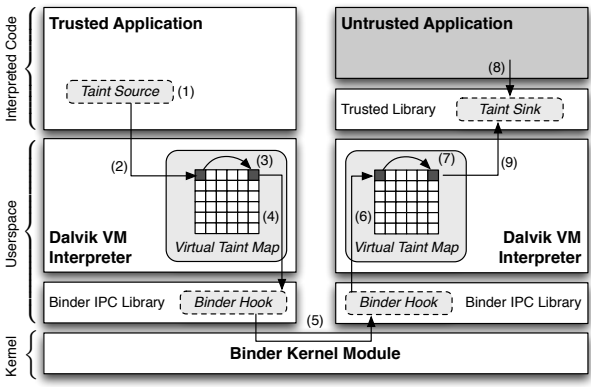

Figure 2: TaintDroid architecture within Android.

quently 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 [32], 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. Note that the Technical Report [17] version of this paper contains more implementation details.

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 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 [57, 7]. Tracked memory is unstructured and without content semantics. Frequently taint tags are stored in non-adjacent shadow memory [57] and tag maps [61]. 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: Dalvik loads local variables into registers for use in methods. Registers contain primitive type values and object references, and are always 32 bits, with *long* and *double* type variables occupying two adjacent registers. The interpreter stores registers on an internal execution state stack. 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 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.

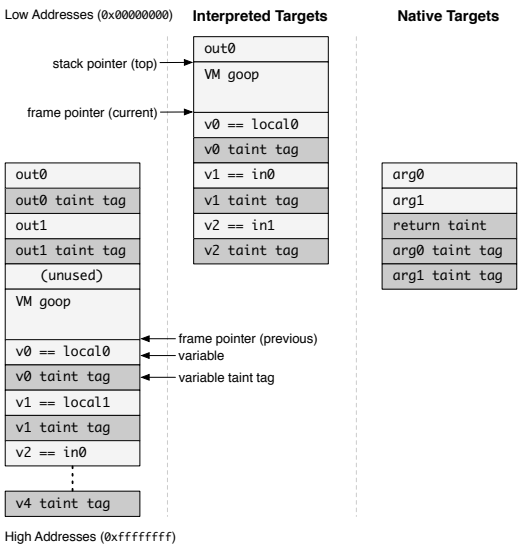

Figure 3: Modified Stack Format. Taint tags are interleaved between registers for interpreted method targets and appended for native methods. Dark grayed boxes represent taint tags.

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 $fp[2 \cdot i + 1]$.

Method Arguments: A target method is either interpreted or native. The Dalvik VM uses the execution state stack to pass arguments to both target types. Before a method is invoked, copies of the specified argument registers are pushed onto the stack (the copies disappear on method termination, which impacts the taint library design, as discussed in Section 4.6). If the target method is interpreted, the new values become high numbered registers in the callee stack frame. That is, interpreted method arguments become local variable registers and hence require consistent taint instrumentation. If the target method is native, a pointer to the stack top is passed to the native method. The target native method receives a pointer to a byte array from which it must parse 32 and 64-bit values in accordance to its method signature.

Figure 3 depicts TaintDroid's method argument modifications for both interpreted and native methods. 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. Therefore, interleaving taint tags for native method arguments would require pervasive modification. Not all native methods require taint tags for correct taint propagation (see Section 4.3). By appending argument taint tags as shown in Figure 3, we maintain compatibility and reduce source code modifications.

Finally, as is discussed in Section 4.3, TaintDroid's native method instrumentation also requires transfer of the return value taint tag. We use the interpreter stack to communicate the return value taint tag to the interpreter. This modification results in an unused 32-bit spacer for interpreted methods. This communication option maintains native method interface compatibility, which simplifies instrumentation.

Class Fields: DEX byte-code maintains Java's class and object semantics. Java defines two types of class fields: static and instance. Static fields store one value per class definition and are shared across all class instances. Instance fields store a different value for each class instance.

Static field storage is straightforward, as values are stored directly in a data structure managed by the interpreter, hence allowing adjacent storage of taint tags. Instance fields require more careful instrumentation. The corresponding data structure does not store values, but rather a byte-offset into a data object instance. Here, we interleave taint tags with values in the class instance data object, causing the taint tag to always exist in the 32-bits following the value. [2]

Arrays: TaintDroid stores one taint tag per array, which incurs significantly less storage than storing one tag per value. Per-value taint tag storage would be severely inefficient for Java *String* objects, as each character would require its own tag and causes string manipulation to copy individual character taint tags.

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 scalar 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; this section addresses why these

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

cases exist. However, first we present taint tracking in the Dalvik machine language as a formal logic.

4.2.1 Taint Propagation Logic

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 [29] (see Section 8). 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 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 variable v_2 to v_1.

Definitions 1-4 provide the primitives required to define runtime taint propagation for Dalvik VM. Table 1 captures the 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, respectively, maintained within the interpreter. A, B, and C are constants in the byte-code.

The taint propagation logic uses conservative data flow 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 byte-code does not require idioms to clear values (e.g., "`xor eax, eax`" in x86), therefore no special handling is required.

Array instructions propagate taint tags to and from array objects (recall that we store one taint tag per array). The *aput-op* instruction taint logic unions the existing array taint tag with the taint tag of the variable to be stored. The *aget-op* instruction logic assigns the destination register the union of the array and index taint tags. Note that this is a deviation from data flow, but is commonly included in taint propagation logic (e.g., in Panorama [57]) to account for translation tables commonly used for character conversion.

DEX also defines several array-related instructions not included in Table 1 (for brevity). The *array-length* instruction returns the length of an array. Some taint propagation logics taint array length to aid direct control flow propagation (e.g., Vogt et al. [53]). We only consider data flow propagation, therefore we assign a taint tag of $\emptyset$. Next, the *new-array* and *fill-new-array* instructions allocate a new array with constant values, therefore we set the new array's taint tag to $\emptyset$. Finally, the *fill-data-array* copies values from the byte-code into an array. We assign a taint tag of $\emptyset$ to the array if none of the original array content remains.

The field *put* and *get* instructions have data flow semantics similar to the move instructions. This intuition holds for static fields and the instance field *put* instruction. However, after investigating Dalvik VM runtime behavior, we determined instance *get* instructions (*iget*) should include the object reference taint tag as described in the next section.

Finally, there are two miscellaneous DEX instruction types not included in Table 1 that do not propagate taint tags but require instrumentation. DEX defines a set of *cmp-X* instructions that perform a comparison between registers and assigns a destination register the value of 0, 1, or -1. As we only consider data flow, we assign the destination register a taint tag of $\emptyset$. Note that propagating taint to the destination register would provide con-

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

```
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); } }
}
```

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

text for direct control flow taint propagation if a "taint scope" were statically extracted from the DEX byte-code before execution. Lastly, the *instance-of* instruction corresponds to the *instanceof* operation in Java. We do not consider an object instance's type as ever containing a taint marking, therefore we set the destination register's taint tag to $\emptyset$.

4.2.2 Tainting Object References

The propagation rules in Table 1 are straightforward with two exceptions. First, taint propagation logics commonly include the taint tag of an array index during lookup to handle translation tables (e.g., ASCII/UNICODE or character case conversion). For example, consider a translation table from lowercase to upper case 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. Therefore, we 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 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 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.

4.3.1 Internal VM Methods

Internal VM method arguments include a pointer to an array of 4-byte values containing Java 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 inspect and instrument Dalvik's internal VM methods for taint propagation. Only a subset of the internal VM methods require modification. For those that do, we manually acquire taint tags appended to the Java argument array and assign a taint tag to the memory slot reserved for the return value taint tag. We also modify the interpreter to copy this value to the internally managed return value taint tag after the method terminates.

We identified 185 internal VM methods in Android version 2.1. This list was further narrowed by considering method names and argument types. We manually inspected and instrumented the remaining methods (if necessary). For example, the *System.arraycopy()* native method copies the contents of one array object to another. Our instrumentation acquires the taint tag stored in the source array and assigns it to the destination array. Several native methods implementing Java reflection also required instrumentation.

4.3.2 JNI Methods

JNI methods are called by a call bridge, which is an internal VM method. The call bridge parses the argument array based on a method descriptor string indicating the number and type of Java arguments. The call bridge then copies the values into the native instruction set calling convention. As a consequence, JNI methods cannot retrieve and return taint tags in the same way performed for internal VM methods. However, the call bridge provides a valuable mediation hook for generic and extensible taint propagation rules. We use a combination of heuristics and method profiles to capture information flow in JNI methods. The heuristic exists to minimize the effort required to define method profiles and is unnecessary, given an automated means of defining the profiles (as described below).

The propagation heuristic provides conservative propagation for JNI methods that only operate on primitive type variables (a common property). The heuristic calculates the union of the taint tags associated with the method arguments and assigns the result to the taint tag of the return value. For example, the *cos()* math library (a JNI method in Android) takes one argument and returns the cosine of that value, where there is a flow from the argument to the return value. 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.

TaintDroid also defines *method profiles*, which are lists of $(from, to)$ pairs indication information flow between variables. The profile may specify method parameters, class variables, and return values. If any of these variables are objects, the profile specifies the object type and allows arbitrary levels of dereferencing by variable name and type. The profile is automatically applied on method termination.

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.[3] 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. Future work will provide a more indepth static analysis to identify flows and automatically generate method profiles. Currently, we define method profiles as needed. For example, methods in the IBM *NativeConverter* class require propagation for conversion between character and byte arrays. These methods are frequently used when transmitting strings over network connections.

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

We modified the C++ parcel message object to store one taint tag per parcel and added two interface methods: *updateTaint()* and *getTaint()*. The former method unions its argument tag value with the existing parcel taint tag. The latter method retrieves parcel's current taint tag. We then modified the Java parcel object with shims for all marshall (e.g., *writeInt()*) and unmarshall (e.g., *readInt()*) methods. The shims use our taint library (Section 4.6) to acquire and set taint tags on Java variables. We modified the Java interface, because the C++ JNI interface cannot access argument taint tags.

The binder IPC mechanism transfers C++ parcel ob-

[3]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.

jects. The IPC transmission passes the byte array maintained by a parcel to the binder kernel module, which copies the memory into the remote process. TaintDroid appends the parcel taint tag to the byte array immediately before transmission to the kernel and sets the taint tag of the parcel object in the remote process upon receipt and requires no kernel modifications.

We chose to implement message-level over variable-level taint propagation, because in a variable-level system, a devious receiver could game the monitoring by unpacking variables in a different way to acquire values without taint propagation. For example, if an IPC parcel message contains a sequence of scalar values, the receiver may unpack a string instead, thereby acquiring values without propagating all the taint tags on scalar values in the sequence. Hence, to prevent applications from removing taint tags in this way, the current implementation protects taint tags at the message-level.

Message-level taint propagation for IPC leads to false positives. Similar to arrays, all data items in a parcel share the same taint tag. For example, Section 8 discusses limitations for tracking the IMSI that results from passing as portions the value as configuration parameters in parcels. Future implementations will consider word-level taint tags along with additional consistency checks to ensure accurate propagation for unpacked variables. However, this additional complexity will negatively impact IPC performance.

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 and limits the granularity of taint markings for information 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. 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.

Taint sources can only add taint tags to memory for which TaintDroid provides tag storage. Currently, taint source and sink placement is limited to variables in interpreted code, IPC messages, and files. 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, 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. 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. This organization provides a useful un-

ambiguous 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. Note that while TaintDroid's 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 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 30 popular third-party Android applications use privacy sensitive user data. Existing applications acquire 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] (1,100 applications, in total) revealed that roughly a third of the applications (358 of the 1,100 applications) require Internet permissions along with permissions to access either location, camera, or audio data. From this set, we randomly selected 30 popular applications (an 8.4% sample size), which span 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 inactivate, 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.

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 [21] and FlurryAgent [20] 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 20 out of the 30 applications require permissions to read phone state and the Internet. We found that 2 of the 20 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.[4] In neither case was the

[4]Because of the limitation of the IMSI taint source as discussed in

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); Yellow Pages (Reference); 3001 Wisdom Quotes Lite, Dastelefonbuch, Astrid (Productivity), BBC News Live Stream (News & Weather); Ringtones (Entertainment)	14	x			x
Layar (Lifestyle); Knocking (Social); Coupons (Shopping); Trapster (Travel); Spongebob Slide (Game); ProBasketBall (Sports)	6	x	x		x
MySpace (Social); Barcode Scanner, ixMAT (Shopping)	3		x		
Evernote (Productivity)	1	x	x	x	

[*] Listed names correspond to the name displayed on the phone and not necessarily the name listed in the Android Market.

[†] 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.

[*] TaintDroid flagged nine applications in this category, but only seven transmitted the raw IMEI without mentioning such practice in the EULA.

[†]To the best of our knowledge, the binary messages contained tainted location data (see the discussion below).

user informed that this information was transmitted 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 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 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 more 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. 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:

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.

```
...&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 and send 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 demonstrates 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 clearly 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.

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

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

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 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.

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

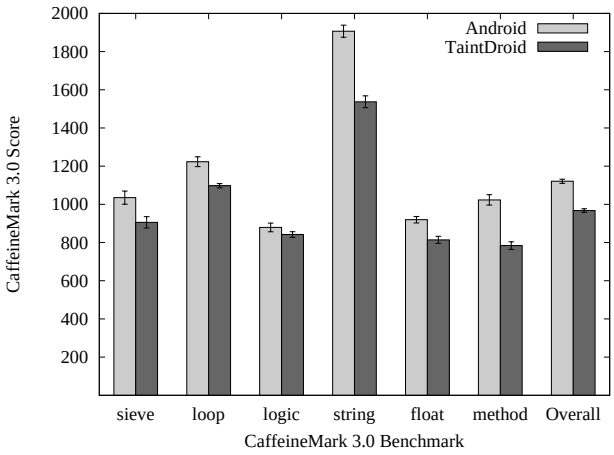

Figure 5: Microbenchmark of Java overhead. Error bars indicate 95% confidence intervals.

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 adds 498 ms to the 1718 ms needed by Android to take a picture (an overhead of 29%). A portion of this overhead can be attributed to to additional file operations required for taint propagation (one *getxattr*/*setxattr* pair per written data buffer). Note that some of this overhead can be reduced by eliminating redundant taint propagation. That is, only the taint tag for the first data buffer written to file needs to be propagated. For example, the current taint tag could be associated with the file descriptor.

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 [43]. 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.

Table 5: IPC Throughput Test (10,000 msgs).

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

This is most likely due to the additional memory comparisons that occur when the JNI propagation heuristic checks for string objects in method prototypes.

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. Note that much of an Android process's memory is used by the zygote runtime environment. These native library memory pages are shared between applications to reduce the overall system memory footprint and require taint tracking. Given that TaintDroid stores 32 taint markings (4 bytes) for each 32-bit variable in the interpreted environment (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 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 applica-

tions' 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, 37], 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 [53]; 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 [39]; 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, 57]. 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 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), which are all tainted together.[5] 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.

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

9 Related Work

Mobile phone host security is a growing concern. OS-level protections such as Kirin [18], Saint [42], and Security-by-Contract [15] provide enhanced security mechanisms for Android and Windows Mobile. These approaches prevent access to sensitive information; however, once information enters the application, no additional mediation occurs. In systems with larger displays, a graphical widget [27] can help users visualize sensor access policies. Mulliner et al. [36] provide information tracking by labeling smartphone processes based on the interfaces they access, effectively limiting access to future interfaces based on acquired labels.

Decentralized information flow control (DIFC) enhanced operating systems such as Asbestos [52] and HiStar [60] label processes and enforce access control based on Denning's lattice model for information flow security [13]. Flume [30] provides similar enhancements for legacy OS abstractions. DEFCon [34] uses a logic similar to these DIFC OSes, but focuses on events and modifies a Java runtime with lightweight isolation. Related to these system-level approaches, PRECIP [54] 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 [28] and TightLip [59]. 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 [46] extends existing programming languages by labeling variables with security attributes. Compilers use the security labels to generate security proofs, e.g., Jif [37, 38] and SLam [24]. Laminar [45] provides DIFC guarantees based on programmer defined security regions. However, these languages require careful development and are often incompatible with legacy software designs [25].

Dynamic taint analysis provides information tracking for legacy programs. The approach has been used to enhance system integrity (e.g., defend against software attacks [41, 44, 8]) and confidentiality (e.g., discover privacy exposure [57, 16, 61]), as well as track Internet worms [9]. Dynamic tracking approaches range from whole-system analysis using hardware extensions [51, 11, 50] and emulation environments [7, 57] to per-process tracking using dynamic binary translation (DBT) [6, 44, 8, 61]. The performance and memory overhead associated with dynamic tracking has resulted in an array of optimizations, including optimizing context switches [44], on-demand tracking [26] based

on hypervisor introspection, and function summaries for code with known information flow properties [61]. If source code is available, significant performance improvements can be achieved by automatically instrumenting legacy programs with dynamic tracking functionality [56, 31]. Automatic instrumentation has also been performed on x86 binaries [47], 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. 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.

Finally, dynamic taint analysis has been applied to virtual machines and interpreters. Haldar et al. [22] instrument the Java String class with taint tracking to prevent SQL injection attacks. WASP [23] 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. [39] instrument the Kaffe JVM. Vogt et al. [53] instrument a Javascript interpreter to prevent cross-site scripting attacks. Xu et al. [56] automatically instrument the PHP interpreter source code with dynamic information tracking to prevent SQL injection attacks. Finally, the Resin [58] environment for PHP and Python uses data flow 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 de-serialize 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 a range of operating system sharing mechanisms.

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 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. Our findings demonstrate the effectiveness and value of enhancing smartphone platforms with monitoring tools such as TaintDroid.

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 paper. We also thank Kevin Butler, Stephen McLaughlin, Machigar Ongtang, and the SIIS lab as a whole for their helpful comments. This material is based upon work supported by the National Science Foundation. William Enck and Patrick McDaniel were partially supported by NSF Grant No. CNS-0905447, CNS-0721579 and CNS-0643907. Landon Cox and Peter Gilbert were partially supported by NSF CAREER Award CNS-0747283. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

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. Apples App Store Downloads Top Three Billion. `http://www.apple.com/pr/library/2010/01/05appstore.html`, January 2010.

[5] CHANDRA, D., AND FRANZ, M. 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] CHENG, W., ZHAO, Q., YU, B., AND HIROSHIGE, S. TaintTrace: Efficient Flow Tracing with Dyanmic Binary Rewriting. In *Proceedings of the IEEE Symposium on Computers and Communications (ISCC)* (June 2006), pp. 749–754.

[7] CHOW, J., PFAFF, B., GARFINKEL, T., CHRISTOPHER, K., AND ROSENBLUM, M. Understanding Data Lifetime via Whole System Simulation. In *Proceedings of the 13th USENIX Security Symposium* (August 2004).

[8] CLAUSE, J., LI, W., AND ORSO, A. Dytan: A Generic Dynamic Taint Analysis Framework. In *Proceedings of the 2007 international symposium on Software testing and analysis* (2007), pp. 196–206.

[9] COSTA, M., CROWCROFT, J., CASTRO, M., ROWSTRON, A., ZHOU, L., ZHANG, L., AND BARHAM, P. Vigilante: End-to-End Containment of Internet Worms. In *Proceedings of the ACM Symposium on Operating Systems Principles* (2005).

[10] COX, L. P., AND GILBERT, P. RedFlag: Reducing Inadvertent Leaks by Personal Machines. Tech. Rep. TR-2009-02, Duke University, 2009.

[11] CRANDALL, J. R., AND CHONG, F. T. Minos: Control Data Attack Prevention Orthogonal to Memory Model. In *Proceedings of the International Symposium on Microarchitecture* (December 2004), pp. 221–232.

[12] DAVIES, C. 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] DENNING, D. E. A Lattice Model of Secure Information Flow. *Communications of the ACM 19*, 5 (May 1976), 236–243.

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

[15] DESMET, L., JOOSEN, W., MASSACCI, F., PHILIPPAERTS, P., PIESSENS, F., SIAHAAN, I., AND VANOVERBERGHE, D. Security-by-contract on the .NET platform. *Information Security Technical Report 13*, 1 (January 2008), 25–32.

[16] EGELE, M., KRUEGEL, C., KIRDA, E., YIN, H., AND SONG, D. Dyanmic Spyware Analysis. In *Proceedings of the USENIX Annual Technical Conference* (June 2007), pp. 233–246.

[17] ENCK, W., GILBERT, P., CHUN, B.-G., COX, L. P., JUNG, J., MCDANIEL, P., AND SHETH, A. N. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. Tech. Rep. NAS-TR-0120-2010, Network and Security Research Center, Department of Computer Science and Engineering, Pennsylvania State University, University Park, PA, USA, August 2010.

[18] ENCK, W., ONGTANG, M., AND MCDANIEL, P. On Lightweight Mobile Phone Application Certification. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS)* (November 2009).

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

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

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

[22] HALDAR, V., CHANDRA, D., AND FRANZ, M. Dynamic Taint Propagation for Java. In *Proceedings of the 21st Annual Computer Security Applications Conference (ACSAC)* (December 2005), pp. 303–311.

[23] HALFOND, W. G., ORSO, A., AND MANOLIOS, P. WASP: Protecting Web Applications Using Positive Tainting and Syntax-Aware Evaluation. *IEEE Transactions on Software Engineering 34*, 1 (2008), 65–81.

[24] HEINTZE, N., AND RIECKE, J. G. The SLam Calculus: Programming with Secrecy and Integrity. In *Proceedings of the Symposium on Principles of Programming Languages (POPL)* (1998), pp. 365–377.

[25] HICKS, B., AHMADIZADEH, K., AND MCDANIEL, P. Understanding practical application development in security-typed languages. In *22st Annual Computer Security Applications Conference (ACSAC)* (2006), pp. 153–164.

[26] HO, A., FETTERMAN, M., CLARK, C., WARFIELD, A., AND HAND, S. Practical Taint-Based Protection using Demand Emulation. In *Proceedings of the European Conference on Computer Systems (EuroSys)* (2006), pp. 29–41.

[27] HOWELL, J., AND SCHECHTER, S. 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).

[28] JUNG, J., SHETH, A., GREENSTEIN, B., WETHERALL, D., MAGANIS, G., AND KOHNO, T. Privacy Oracle: A System for Finding Application Leaks with Black Box Differential Testing. In *Proceedings of ACM CCS* (2008).

[29] KING, D., HICKS, B., HICKS, M., AND JAEGER, T. Implicit Flows: Can't Live with 'Em, Can't Live without 'Em. In *Proceedings of the International Conference on Information Systems Security* (2008).

[30] KROHN, M., YIP, A., BRODSKY, M., CLIFFER, N., KAASHOEK, M. F., KOHLER, E., AND MORRIS, R. Information Flow Control for Standard OS Abstractions. In *Proceedings of ACM Symposium on Operating Systems Principles* (2007).

[31] LAM, L. C., AND CKER CHIUEH, T. A General Dynamic Information Flow Tracking Framework for Security Applications. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)* (2006).

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

[33] LOOKOUT. Introducing the App Genome Project. http://blog.mylookout.com/2010/07/introducing-the-app-genome-project/, July 2010.

[34] MIGLIAVACCA, M., PAPAGIANNIS, I., EYERS, D. M., SHAND, B., BACON, J., AND PIETZUCH, P. DEFCon: High-Performance Event Processing with Information Security. In *PROCEEDINGS of the USENIX Annual Technical Conference* (2010).

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

[36] MULLINER, C., VIGNA, G., DAGON, D., AND LEE, W. Using Labeling to Prevent Cross-Service Attacks Against Smart Phones. In *Proceedings of Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)* (2006).

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

[38] MYERS, A. C., AND LISKOV, B. Protecting Privacy Using the Decentralized Label Model. *ACM Transactions on Software Engineering and Methodology 9*, 4 (October 2000), 410–442.

[39] NAIR, S. K., SIMPSON, P. N., CRISPO, B., AND TANENBAUM, A. S. 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).

[40] NEWSOME, J., MCCAMANT, S., AND SONG, D. Measuring channel capacity to distinguish undue influence. In *ACM SIGPLAN Workshop on Programming Languages and Analysis for Security* (2009).

[41] NEWSOME, J., AND SONG, D. 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).

[42] ONGTANG, M., MCLAUGHLIN, S., ENCK, W., AND MCDANIEL, P. Semantically Rich Application-Centric Security in Android. In *Proceedings of the 25th Annual Computer Security Applications Conference (ACSAC)* (2009).

[43] PENDRAGON SOFTWARE CORPORATION. CaffeineMark 3.0. http://www.benchmarkhq.ru/cm30/.

[44] QIN, F., WANG, C., LI, Z., SEOP KIM, H., ZHOU, Y., AND WU, Y. 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* (2006), pp. 135–148.

[45] ROY, I., PORTER, D. E., BOND, M. D., MCKINLEY, K. S., AND WITCHEL, E. Laminar: Practical Fine-Grained Decentralized Information Flow Control. In *Proceedings of Programming Language Design and Implementation* (2009).

[46] SABELFELD, A., AND MYERS, A. C. Language-based information-flow security. *IEEE Journal on Selected Areas in Communication 21*, 1 (January 2003), 5–19.

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

[48] SCHWARTZ, E. J., AVGERINOS, T., AND BRUMLEY, D. 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).

[49] SLOWINSKA, A., AND BOS, H. Pointless Tainting? Evaluating the Practicality of Pointer Tainting. In *Proceedings of the European Conference on Computer Systems (EuroSys)* (April 2009), pp. 61–74.

[50] SUH, G. E., LEE, J. W., ZHANG, D., AND DEVADAS, S. Secure Program Execution via Dynamic Information Flow Tracking. In *Proceedings of Architectural Support for Programming Languages and Operating Systems* (2004).

[51] VACHHARAJANI, N., BRIDGES, M. J., CHANG, J., RANGAN, R., OTTONI, G., BLOME, J. A., REIS, G. A., VACHHARAJANI, M., AND AUGUST, D. I. RIFLE: An Architectural Framework for User-Centric Information-Flow Security. In *Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture* (2004), pp. 243–254.

[52] VANDEBOGART, S., EFSTATHOPOULOS, P., KOHLER, E., KROHN, M., FREY, C., ZIEGLER, D., KAASHOEK, F., MORRIS, R., AND MAZIÈRES, D. Labels and Event Processes in the Asbestos Operating System. *ACM Transactions on Computer Systems (TOCS) 25*, 4 (December 2007).

[53] VOGT, P., NENTWICH, F., JOVANOVIC, N., KIRDA, E., KRUEGEL, C., AND VIGNA, G. Cross-Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In *Proc. of Network & Distributed System Security* (2007).

[54] WANG, X., LI, Z., LI, N., AND CHOI, J. Y. PRECIP: Towards Practical and Retrofittable Confidential Information Protection. In *Proceedings of 15th Network and Distributed System Security Symposium (NDSS)* (2008).

[55] WhatApp. http://www.whatapp.org. Accessed April 2010.

[56] XU, W., BHATKAR, S., AND SEKAR, R. Taint-Enhanced Policy Enforcement: A Practical Approach to Defeat a Wide Range of Attacks. In *Proceedings of the USENIX Security Symposium* (August 2006), pp. 121–136.

[57] YIN, H., SONG, D., EGELE, M., KRUEGEL, C., AND KIRDA, E. Panorama: Capturing System-wide Information Flow for Malware Detection and Analysis. In *Proceedings of ACM Computer and Communications Security* (2007).

[58] YIP, A., WANG, X., ZELDOVICH, N., AND KAASHOEK, M. F. Improving Application Security with Data Flow Assertions. In *Proceedings of the ACM Symposium on Operating Systems Principles* (Oct. 2009).

[59] YUMEREFENDI, A. R., MICKLE, B., AND COX, L. P. TightLip: Keeping Applications from Spilling the Beans. In *Proceedings of the 4th USENIX Symposium on Network Systems Design & Implementation (NSDI)* (2007).

[60] ZELDOVICH, N., BOYD-WICKIZER, S., KOHLER, E., AND MAZIÈRES, D. Making Information Flow Explicit in HiStar. In *Proceedings of the 7th symposium on Operating Systems Design and Implementation (OSDI)* (2006).

[61] ZHU, D., JUNG, J., SONG, D., KOHNO, T., AND WETHERALL, D. Privacy Scope: A Precise Information Flow Tracking System For Finding Application Leaks. Tech. Rep. EECS-2009-145, Department of Computer Science, UC Berkeley, 2009.