DOI:10.1145/2494522

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

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

Abstract

Today's smartphone operating systems frequently fail to provide users with adequate control over and visibility into how third-party applications use their privacy-sensitive data. We address these shortcomings with TaintDroid, an efficient, systemwide dynamic taint tracking and analysis system capable of simultaneously tracking multiple sources of sensitive data. TaintDroid provides real-time analysis by leveraging Android's virtualized execution environment. Using TaintDroid to monitor the behavior of 30 popular third-party Android applications, we found 68 instances of misappropriation of users' location and device identification 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. 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.

Resolving the tension between the fun and utility provided by third-party applications and the privacy risks they pose is a critical challenge for smartphone platforms. Smartphone 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 problem inherently cannot be solved by traditional access control techniques. The naïve solution is to disallow network access once privacy-sensitive information is received by a process. However, perhaps more so than other platforms, smartphone applications are built around cloud services. This has two major implications. First, the vast majority of applications functionally require network access. Second, and perhaps more important, some applications must send privacy-sensitive information to specific network hosts to meet the needs of the user. Therefore, the problem is not simply one of determining *if* such information is sent to the network, but rather to determine *what* information is sent *where*.

Determining how an application uses and discloses privacy-sensitive information is achievable using fine-grained dynamic taint analysis, commonly known as "taint tracking." A "taint" is simply a label on a data item or variable. The label assigns a semantic type (e.g., geographic location) to the data, and may simultaneously encode multiple such types (commonly called a *taint tag*). It is the task of the taint tracking system to (1) assign taint labels at a *taint source*, (2) automatically propagate taint labels to dependent data and variables, and finally (3) take some action based on the taint label of data at a *taint sink*. For example, a taint tracking system might label the variables containing the geographic coordinates of a phone when they are returned from the location API (taint source), propagate that label to all variables that are derived from those variables (e.g., if `a = b + c`, then `a` is derived from `b` and `c`), and then take some action (e.g., log and drop) when a variable with a location label reaches the network API (taint sink).

Security literature has many examples of taint tracking, but proposed solutions are either coarse or slow. We show that taint tracking can be efficient for Android applications. Furthermore, we find that monitoring only a single process is insufficient in Android, as data commonly flows between applications.

The original version of this paper was published in the *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, 2010.

Our goal is to create a whole-system taint tracking framework that operates in *real time* to detect sensitive data exposure with sufficient context to identify potentially misbehaving applications. The real-time constraint enables both daily use by concerned users and efficient study by external security services. As with any such practical system, the solution design requires careful trade-offs between performance and precision.

We introduce the TaintDroid extension to Android, which is the first practical system that can track the flow of privacy-sensitive data throughout a smartphone platform. To balance performance and tracking precision, TaintDroid leverages 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 appropriate trade-offs between tracking precision and performance for resource-constrained smartphones.

Our second contribution lies in our use of TaintDroid to perform the first study of smartphone applications that identifies extensive misuse of privacy-sensitive data. This study considered 30 randomly selected, popular Android applications that use location, camera, or microphone data. These applications were manually run on a phone with a TaintDroid firmware. We then collected various TaintDroid and network logs and noted user expectations of privacy-sensitive data exposure to evaluate potential data misuse. In our experiments, TaintDroid correctly flagged 105 TCP connections as containing privacy-sensitive information. After further inspection, 37 were classified as clearly legitimate. By inspecting the remaining 68 TCP connections, we discovered 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. These results raise strong concerns of potential widespread collection of geographic location and phone identifiers without users' knowledge.

2. DESIGN OVERVIEW

We seek a design that allows users to monitor how third-party smartphone applications handle their private data in real time. Existing static analysis techniques that require source code are not suitable as many smartphone applications are closed-source and techniques that convert bytecode to source code are still far from error-free. Even if source code is available, runtime events and configuration often dictate information use; real-time monitoring accounts for these environment-specific dependencies. Furthermore, we assume that all downloaded third-party applications are untrusted and that these applications run simultaneously.

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

- *Smartphones are resource constrained*. The resource limitations of smartphones preclude the use of heavy-weight information tracking systems.
- *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.
- *Privacy-sensitive information 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 Inter Process Communication (IPC) between applications, including core system applications designed to disseminate privacy-sensitive information.

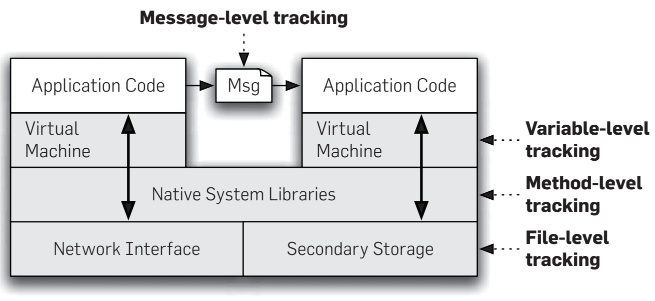
Whole-system, fine-grained dynamic taint analysis satisfies these challenges, if performance limitations and overtainting can be overcome. Here, sensitive information is first identified at a *taint source* that assigns a *taint marking* indicating the information type. Smartphones have well-defined application programming interfaces (APIs) for retrieving privacy-sensitive information (e.g., microphone, location, and phone identifiers). The dynamic taint analysis then tracks how the labeled data impacts other data in a way that might leak the original sensitive information. This tracking is often performed at the instruction level. For example, if the instruction `a = b + c` is executed and `c` has a taint t, `a` will have taint t after the instruction has executed. Finally, the impacted data is identified before it leaves the system at a *taint sink* (the network interface in our design).

Clearly, performing dynamic taint analysis at the instruction level incurs significant performance overhead. For example, whole-system emulation and per-process dynamic binary translation (DBT)[2, 4, 17] commonly incur 2–20 times slowdown. This overhead occurs because for each monitored instruction, the tracking framework must (1) save the execution context, (2) perform the taint propagation, and then (3) restore the execution context.

We overcome this limitation by moving the tracking framework inside of the OS and taking advantage of the virtual machine (VM)-based architecture used by Android, BlackBerry, and Windows Phone. For these platforms, applications consist of Java-based or .NET byte-code that is executed within an interpreter. We modify the interpreter to perform taint propagation and then carefully extend propagation to the rest of the system. By modifying the interpreter, we avoid saving and restoring execution context. Furthermore, our approach focuses on tracking the data used by interpreted code, which is only a small fraction of the overall process memory. Note that our approach is not compatible with iOS, since it uses binary applications.

Figure 1 presents our tracking design. Tracking occurs in four ways. First, we instrument the VM interpreter to provide *variable-level tracking* within untrusted application code. Using variable semantics increases precision over traditional x86 tracking logics and focuses taint marking storage on data instead of code. Second, we use *message-level tracking* between applications. The message granularity minimizes IPC overhead while extending the analysis system-wide. Third, for system-provided native libraries, we use *method-level tracking*. Here, we run platform native code without instrumentation and patch the taint propagation on return. Finally, we

Figure 1. Multi-level approach for performance-efficient taint tracking within Android.



use *file-level* tracking to ensure that persistent information conservatively retains its taint markings.

While this design allows practical real-time tracking, it relies on the firmware's integrity. We trust the virtual machine executing in user space and any native system libraries loaded by the untrusted interpreted application. Hence, we assume that only platform native libraries can be loaded. Without this, applications can not only remove taint markings, but also corrupt the tracking within the interpreter. In our target platform (Android), we modified the native library loader to only load native libraries from the firmware. To test compatibility, we surveyed the top 50 most popular free applications in each category of the Android Market (1,100 applications in total) in July 2010 and found that less than 5% of applications included a `.so` file. Therefore, we expect that TaintDroid is incompatible with only a small percentage of applications.

3. TAINTDROID

TaintDroid is a realization of our multiple granularity taint tracking for Android. Central to the design is a careful trade-off between tracking precision and performance. 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 propagation is extended to IPC and files.

This section overviews the core implementation challenges of TaintDroid. Here we discuss (a) taint tag storage, (b) interpreted code taint propagation, (c) native code taint propagation, (d) IPC taint propagation, and (e) secondary storage taint propagation. Additional details can be found in our original paper.[9]

3.1. Taint tag storage

Taint tag storage impacts both performance and memory overhead. Traditional taint tracking systems store one tag for every data byte or word.[3, 23] Often, this tag consists of a single bit in implementations. To further reduce storage overhead, such systems only maintain tags for tainted bytes using non-adjacent shadow memory[23] or tag maps.[25] TaintDroid takes a different approach. Since we know which bytes are variables, we significantly reduce the scope of memory to track by only keeping track of the taint states of variables. This allows TaintDroid to store taint tags adjacent to variables in memory, which provides spatial locality when accessing taint tags at runtime. Furthermore, it allows one to practically store a 32-bit bit vector with each variable, allowing 32 different taint markings.

TaintDroid adds taint tag storage for all scalar values in Android's Dalvik VM interpreter. Android applications are written in Java, but compiled to a special DEX bytecode that is executed by Dalvik. Given these Java origins, TaintDroid must provide taint tag storage for method local variables, method arguments, class static fields, class instance fields, and arrays.

DEX bytecode differs from Java bytecode in that it is register based. This is important to the TaintDroid implementation. When a DEX method is called, Dalvik creates a new stack frame that allocates 32-bit register storage for all of the scalar and object reference variables used by the method. As shown in Figure 2, method arguments are also stored on the stack and are mapped to high indexed registers in the callee stack frame. TaintDroid provides taint tag storage for these variables by interleaving taint tags between the registers.

TaintDroid stores taint tags adjacent to class fields and arrays within internal data structures. Only one taint tag is stored per array to minimize storage overhead, which is often sufficient for strings. However, this loss in precision may result in false positives. For example, as soon as a tainted value is stored to an array, all values read out of the array will also be tainted. Fortunately, Java arrays frequently contain object references, which are infrequently tainted, resulting in fewer false positives in practice.

3.2. Interpreted code taint propagation

Operating on DEX bytecode provides TaintDroid several distinct advantages. First, all operations have clear semantics. Unlike x86, there is no lack of registers or strange conventions for clearing variables (e.g., `xor %eax, %eax`). Second, scalar values are distinct from pointers. This allows taint propagation to be more precise. Finally, variables that are not method local have clear taint tag storage (described above) that retains types.

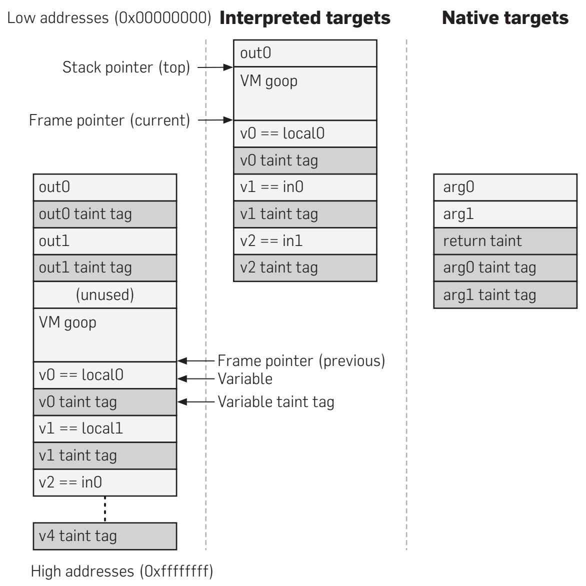
For the most part, taint tag propagation proceeds as one might expect. Instructions always overwrite the destination register; therefore, unary operations set the taint tag of the destination register to that of the source register, and binary operations (e.g., $a = b + c$) set the taint tag of the destination register to the union of the taint tags of the two source registers (e.g., $\tau(a) \leftarrow \tau(b) \cup \tau(c)$). For the implementation, the union is simply a bitwise OR of the taint tag bit vectors. However, there are several cases where the taint propagation is not straightforward (e.g., for array indexes and object references). A full propagation logic and discussion is provided in our original paper.[9]

3.3. Native code taint propagation

Native code is unmonitored in TaintDroid, as performing automated taint propagation would require heavyweight techniques such as dynamic binary translation (DBT). Instead, we synthesize the taint status after the method terminates based on a combination of source code inspection and simple heuristics.

Internal VM methods. The Dalvik VM contains a set of core methods that are called directly by interpreted code and are passed a pointer to an array of 32-bit register

Figure 2. Modified stack format. Taint tags are interleaved between registers for interpreted method targets and appended for native methods. Dark grayed boxes represent taint tags.



arguments and a pointer to a return value. TaintDroid places all the taint tags after the argument values (recall the stack in Figure 2). This ensures that methods that do not impact taint propagation require no modifications. For those that do, the respective taint tags are readily available. Of the 185 internal VM methods in Android version 2.1, only 5 required patching (e.g., for array manipulation and reflection).

JNI methods. The remaining majority of native methods use the Java Native Interface (JNI) and are invoked through a JNI call bridge. The call bridge parses Java arguments and assigns a return value, making it the ideal place to patch the tracking state after a native method executes. To do this, we define a method profile table that defines a list of (*from, to*) pairs indicating flows between method parameters, class variables, and return values. Completely populating the method profile table is best completed using automated static analysis tools; however, for the purposes of this work, we manually defined several methods as needed. To supplement this manual specification, we created a propagation heuristic: *assign the union of the method argument taint tags to the taint tag of the return value*. This heuristic is conservative if the method only operates on primitive and string arguments and return values. For Android version 2.1, we found this condition to hold for 913 of the 2,844 JNI methods. The remaining methods may have false negatives and potentially require explicit method profile specification. While we found these methods effective for our investigations, more thorough consideration of native code is a valuable direction for future work.

3.4. IPC taint propagation

When Android applications communicate with one another, they send *parcel* objects over the binder IPC interface. It is important for TaintDroid to propagate taint tags on parcels to track sensitive information passed not only between downloaded third-party applications, but also between third-party applications and the system. In fact, much of Android's core functionality is implemented using the same application abstractions as third-party software.

TaintDroid assigns one taint tag per parcel message. This results in better performance and lower memory overhead than variable-level or byte-level tracking in parcels. Furthermore, variable-level tracking is subject to manipulation, because the parcel packing of different sized variables is defined by the sender and receiver. However, the disadvantage is false positives (similar to arrays). As we discuss in Section 7, this makes certain taint sources problematic for TaintDroid. Future implementations will investigate the overhead of finer-grained parcel tracking.

3.5. Secondary storage taint propagation

TaintDroid must ensure that when tainted data is written to a file, the taint tag is restored when it is later read. We currently store one taint tag per file, because finer-grained tracking would incur significant overhead. However, the drawback is false positives if the type of tracked information is frequently mixed. In our experiments, this was not a significant problem. To store taint tags, TaintDroid uses extended attributes in the file system. When TaintDroid was developed, the predominately used YAFFS2 file system did not have xattr support, which we needed to add. Official xattr support was later added to YAFFS2, and newer phones have a hardware flash translation layer that allows standard ext4 file systems. A second limitation of the Android storage architecture is the SDcard. Android uses a FAT file system for the SDcard, which does not support xattrs. We formatted the SDcard ext2 and patched the file write API to use file permissions consistent with FAT to ensure compatibility with existing applications.

4. PRIVACY HOOK PLACEMENT

Before TaintDroid can be used to monitor applications, taint sources must be added to the Android Framework. We modified the Android system code to add taint tags to various taint sources. For the most part, we chose to add the taint sources within the Java portion of system applications that retrieve the values from hardware. The following describes the most important classes of taint sources we encountered.

Low-bandwidth sensors. A variety of privacy-sensitive information types are acquired through low-bandwidth sensors, for example, location and accelerometer. Such information often changes frequently and is simultaneously used by multiple applications. Therefore, Android multiplexes access to low-bandwidth sensors using a sensor manager. This sensor manager represents an ideal point for taint source hook placement. We placed hooks in Android's LocationManager and SensorManager applications.

High-bandwidth sensors. Sources such as the microphone and camera are high-bandwidth. Each request from the sensor returns a large amount of data that is only used by one application. Therefore, the OS makes sensor information available via large data buffers, files, or both. When sensor information is shared via files, the file must be tainted with the appropriate tag. We added hooks for both types of API abstractions provided for accessing microphone and camera interfaces.

Information databases. Shared information such as address books and SMS messages are often stored in file-based databases. By adding a taint tag to such database files, all information read from the file will be automatically tainted. We initially used this technique for tracking address book information. Later implementations modified Android's content resolver class to add an appropriate taint tag based on the name of the content provider (i.e., the "authority string") specified by the querying application.

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 7.

Network taint sink. TaintDroid identifies when tainted information is transmitted out the network interface. Our interpreter-based approach requires TaintDroid's code to detect network transmission within interpreted code. Hence, we instrumented the Java framework libraries at the point the native socket library is invoked.

5. APPLICATION STUDY

To demonstrate the utility of TaintDroid, we studied 30 popular third-party Android applications that have access to privacy-sensitive user data and the Internet. This set of applications was randomly selected from a larger set of popular applications that have access to the Internet and to at least one of location, camera, or audio data. We chose to bias our random selection toward applications with access to interesting privacy-sensitive information, because applications without access clearly cannot expose data. The details of our experimental methodology can be found in our original paper.[9] The following describes our major findings.

Our experiments consisted of manually running and exploring the functionality of the applications. We recorded TaintDroid logs and a `tcpdump` packet trace for ground truth. We also took note of End User License Agreements (EULAs) and implicit expectations of data exposure. Our experiments generated 1,130 TCP connections, and TaintDroid correctly flagged 105 TCP connections as containing tainted privacy-sensitive information (i.e., TaintDroid had no false positives). The flagged TCP connections included both plaintext and binary encoded data.

Upon inspecting the 105 flagged TCP connections containing privacy-sensitive information, we found that 37 were for clearly legitimate uses. For example, several of these flagged TCP connections contained HTTP headers indicating the use of the Google Maps for Mobile (GMM) API, and the corresponding application showed a map of the user's location. However, the privacy-sensitive information disclosures in the remaining 68 flagged TCP connections were not expected. These findings are summarized in Table 1.

Location data to advertisement servers. Half of the studied applications exposed location data to third-party advertisement servers without implicit or explicit user consent. Of these 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:

```
...& s=a14a4a93f1e4c68 &..& t=062A1CB1D476DE85B717D
9195A6722A9&d%5Bcoord%5D=47.661227890000006%2C−
122.31589477 &...
```

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

For binary data sent by FlurryAgent, we confirmed location exposure based on 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's log to Android's logcat reports "sending report to http://data.flurry.com/aar.do," which occurs immediately after receiving the tainted parcel.

Our experiments indicate that 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 Table 1) only transmitted location data per user's request to pull localized content from their servers. This finding demonstrates the importance of monitoring how the application *actually* uses or abuses the granted permissions.

Phone information. Of the 30 studied applications, 20 require permissions to read phone state and access 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 that messages were flagged correctly by inspecting the plaintext payload. In neither case was the user informed that this information was transmitted off the phone. Note that while we did not explicitly track the IMSI (see Section 7), it was contained in the plaintext network buffer flagged by TaintDroid.

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 one application that 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

Table 1. Application study results.

	Permissions				Info sent		
Application [package.name]	**Location**	**Phone state**	**Camera**	**Microphone**	**Location**	**Phone info**	**IMEI**
Babble Book [com.kalicinscy.babble]	✓						
Cestos Full [com.chickenbrickstudios.cestos_full]	✓						
Manga Browser [com.mangabrowser.main]	✓				•		
Movies and showtimes [com.stylem.movies]	✓						
Solitare Free [com.mediafill.solitaire]	✓				•		
The Weather Channel [com.weather.Weather]	✓						
3001 Wisdom Quotes Lite [com.xim.wq_lite]	✓	✓			•		
Antivirus Free [com.antivirus]	✓	✓			•	•	•
Astrid [com.timsu.astrid]	✓	✓			•		
BBC News listen & tweet [daaps.media.bbc]	✓	✓			•		
Blackjack [spr.casino]	✓	✓			•		
Bump [com.bumptech.bumpga]	✓	✓					•
Children's ABC Animals (lite) [com.mob4.childrenabc.animals]	✓	✓			•		
Hearts (Free) [com.bytesequencing.hearts_ads]	✓	✓			•		
Horoscope [fr.telemaque.horoscope]	✓	✓			•		•
Mabilo Ringtones [mabilo.ringtones]	✓	✓			•		
The directory for Germany [de.dastelefonbuch.android]	✓	✓					
Traffic Jam Free [com.jiuzhangtech.rushhour]	✓	✓			•		
Wertago for Nightlife [com.wertago]	✓	✓			•		•[†]
Yellow Pages [com.avantar.yp]	✓	✓					•
Knocking Live Video Beta [com.pointyheadsllc.knockingvideo]	✓	✓	✓				•
Layar [com.layar]	✓	✓	✓				○
Pro Basketball Scores [com.plusmo.probasketballscores]	✓	✓	✓		•		
Slide: Spongebob [com.mob4.slideme.qw.android.spongebob]	✓	✓	✓		•		
The coupons App [thecouponsapp.coupon]	✓	✓	✓			•	•
Trapster [com.trapster.android]	✓	✓	✓				•
Barcode Scanner [com.google.zxing.client.android]			✓				
iXmat Barcode Scanner [com.ixellence.ixmat.android.community]			✓				
Myspace [com.myspace.android]			✓				
Evernote [com.evernote]	✓		✓	✓			

✓ = Potential violation; ○ = Clearly stated in EULA; † Sent the hash of the value.

data immediately after it is installed, which is before it is even used.

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 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. 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. Hashing the IMEI provides more protection, because it cannot be reversed to obtain the actual IMEI. However, if all applications hash the IMEI directly, similar privacy concerns can result.

The collection of phone identifiers allows third parties to track user behavior. Phone numbers are often easy to correlate with the owner's name, as many users post their phone number on social networking and other websites. However, collecting seemingly unidentifiable numbers such as the IMSI, ICC-ID, and the IMEI also has privacy implications. First, all applications on the phone use the same phone identifiers. If identifiers and behaviors are collected by an entity that is associated with many applications (e.g., an ad or analytics service), more accurate user profiles can be created. Second, these identifiers are fixed for the duration the user uses the phone, and potentially longer if the SIM card is moved to a new phone. This property means that users cannot simply clear the tracking cookies as they might in a Web browser. Finally, these identifiers are often collected along with personally identifiable information such as email addresses. Such collections create small databases that can be used to correlate actual users with their phone identifiers. Traditionally, this mapping is only held by cellular providers.

6. PERFORMANCE EVALUATION

During the application study, we noticed very little performance overhead. This is likely 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.

We evaluated the performance of TaintDroid for Android

Table 2. Macrobenchmark results.

	Android (ms)	TaintDroid (ms)
App load time	63	65
Address book (create)	348	367
Address book (read)	101	119
Phone call	96	106
Take picture	1718	2216

version 2.1 using macrobenchmarks representing common smartphone activities: loading an application, accessing the address book, making a phone call, and taking a picture. As shown in Table 2, our macrobenchmarks observed negligible overhead (less than 30 ms), with the exception of taking a picture, which added just over half a second. This overhead is likely due to the current method of propagating taint tags to files using xattrs, which could be improved with caching.

While the macrobenchmarks report the performance overhead perceived by users during common smartphone use, we also performed a microbenchmark on Java operations. For this experiment, we used an Android port of the standard CaffeineMark 3.0 benchmark for Java. TaintDroid has an average overall CPU overhead of 14%. We also measured the memory consumption of the benchmark process during the experiments. The benchmark process consumed 21.28MB on Android and 22.21MB on TaintDroid, indicating a 4.4% memory overhead.

7. DISCUSSION

Approach limitations. To minimize performance overhead, TaintDroid only tracks data flows (i.e., explicit flows) and does not track control flows (i.e., implicit flows). Section 5 shows that TaintDroid can track the flow of sensitive data and identify many applications that exfiltrate sensitive information. 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,[7, 14] which is challenging for third-party applications whose source code is unavailable. Direct control flows can be tracked dynamically if a taint scope can be determined[21]; 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[15]; 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.[3, 23] Finally, once information leaves the phone, it may return in a network reply. TaintDroid cannot track such information propagation once the information leaves the phone.

Implementation limitations. Android uses the Apache Harmony 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 occurs 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. Android uses the MCC and MNC extensively as configuration parameters when communicating other data. If the IMSI is treated as tainted, 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 would be more appropriate. However, as our analysis results in Section 5 show, message-level taint tracking is effective for the majority of our taint sources.

8. RELATED WORK

Information flow tracking and control has been the basis of many operating system and programming language designs over the past several decades. For brevity, we focus on systems using dynamic taint analysis, which is primarily used to track information flows in legacy programs. It has been used to enhance system integrity (e.g., defend against software attacks[4, 16, 17]) and confidentiality (e.g., discover privacy exposure[8, 23, 25]), as well as track Internet worms.[5] Dynamic tracking approaches range from whole-system analysis using hardware extensions[6, 19, 20] and emulation environments[3, 23] to per-process tracking using dynamic binary translation (DBT).[2, 4, 17, 25] The performance and memory overhead associated with dynamic tracking have stimulated much research on optimizations, including optimizing context switches,[17] on-demand tracking[12] based on hypervisor introspection, and function summaries for code with known information flow properties.[25] If source code is available, significant performance improvements can be achieved by automatically instrumenting legacy programs with dynamic tracking functionality.[13, 22] Automatic instrumentation has also been performed on x86 binaries,[18] providing a compromise between source code translation and DBT. Our TaintDroid design was inspired by these prior works, but addresses 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.[10] instrument the Java String class with taint tracking to prevent SQL injection attacks. WASP[11] has similar motivations; however, it uses positive tainting of individual characters to ensure that the SQL query contains only high-integrity substrings. Chandra and Franz[1] propose fine-grained information flow tracking within the JVM and instrument Java byte-code to aid control flow analysis. Similarly, Nair et al.[15] instrument the Kaffe JVM. Vogt et al.[21] instrument a Javascript interpreter to prevent cross-site scripting attacks. Xu et al.[22] automatically instrument the PHP interpreter source code with dynamic information tracking to prevent SQL injection attacks. Finally, the Resin[24] 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.

9. CONCLUSION

While smartphone operating systems allow users to control applications' access to sensitive information, users lack visibility into how applications use their private data. To address this, we presented 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, which is achieved by integrating four granularities of taint propagation (variable-level, message-level, method-level, and file-level). Our evaluation shows that TaintDroid has only a 14% performance overhead on a CPU-bound microbenchmark. Previously, most work on taint tracking was either slow (requiring multiple times performance overhead) or required source code. The source code for Android applications is not available; therefore, one might have expected TaintDroid to be very slow. TaintDroid shows this is not the case: one can track information flows of Android applications without source code, with modest overhead.

We used TaintDroid to study the behavior of 30 popular third-party applications and found that two-thirds handle sensitive data inappropriately. In particular, 15 of the 30 applications shared users' locations with remote advertising and analytics servers. Our findings demonstrate the effectiveness and value of enhancing smartphone platforms with monitoring tools such as TaintDroid.

TaintDroid is an ongoing effort that has been incorporated into further projects by both the authors and others in the research community. TaintDroid is available for Android version 2.1, version 2.3 (and adding JIT support), and version 4.1. Information for downloading and building TaintDroid can be found at http://www.appanalysis.org.

Acknowledgments

We thank everyone who helped with the original paper.[9] Enck and McDaniel were partially supported by NSF Grants CNS-0905447, CNS-0721579, and CNS-0643907. Cox and Gilbert were partially supported by NSF CAREER Award CNS-0747283.

References

1. Chandra, D., 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)* (Dec. 2007).
2. Cheng, W., Zhao, Q., Yu, B., Hiroshige, S. TaintTrace: efficient flow tracing with dyanmic binary rewriting. In *Proceedings of the IEEE Symposium on Computers and Communications (ISCC)* (Jun. 2006), 749–754.
3. Chow, J., Pfaff, B., Garfinkel, T., Christopher, K., Rosenblum, M. Understanding data lifetime via whole system simulation. In *Proceedings of the 13th USENIX Security Symposium* (Aug. 2004).
4. Clause, J., Li, W., Orso, A. Dytan: a generic dynamic taint analysis framework. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis* (2007), 196–206.
5. Costa, M., Crowcroft, J., Castro, M., Rowstron, A., Zhou, L., Zhang, L., Barham, P. Vigilante: end-to-end containment of internet worms. In *Proceedings of the ACM Symposium on Operating Systems Principles* (Oct. 2005), 133–147.
6. Crandall, J.R., Chong, F.T. Minos: control data attack prevention orthogonal to memory model. In *Proceedings of the International Symposium on Microarchitecture* (Dec. 2004), 221–232.
7. Denning, D.E., Denning, P.J. Certification of Programs for Secure Information Flow. *Commun. ACM 20*, 7 (Jul. 1977).
8. Egele, M., Kruegel, C., Kirda, E., Yin, H., Song, D. Dyanmic spyware analysis. In *Proceedings of the USENIX Annual Technical Conference* (Jun. 2007), 233–246.
9. Enck, W., Gilbert, P., Chun, B.G., Cox, L.P., Jung, J., McDaniel, P., Sheth, A.N. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Oct. 2010).
10. Haldar, V., Chandra, D., Franz, M. Dynamic taint propagation for Java. In *Proceedings of the 21st Annual Computer Security Applications Conference (ACSAC)* (Dec. 2005), 303–311.
11. Halfond, W.G., Orso, A., Manolios, P. WASP: protecting web applications using positive tainting and syntax-aware evaluation. *IEEE Trans. Softw. Eng. 34*, 1 (2008), 65–81.
12. Ho, A., Fetterman, M., Clark, C., Warfield, A., Hand, S. Practical taint-based protection using demand emulation. In *Proceedings of the European Conference on Computer Systems (EuroSys)* (Apr. 2006), 29–41.
13. Lam, L.C., cker Chiueh, T. A general dynamic information flow tracking framework for security applications. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)* (Dec. 2006), 463–472.
14. Myers, A.C. JFlow: practical mostly-static information flow control. In *Proceedings of the ACM Symposium on Principles of Programming Langauges (POPL)* (Jan. 1999).
15. Nair, S.K., Simpson, P.N., Crispo, B., 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).
16. Newsome, J., Song, D. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the 12th Network and Distributed System Security Symposium (NDSS)* (2005).
17. Qin, F., Wang, C., Li, Z., seop Kim, H., Zhou, Y., 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), 135–148.
18. Saxena, P., Sekar, R., 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)* (Apr. 2008), 74–83,.
19. Suh, G.E., Lee, J.W., Zhang, D., Devadas, S. Secure program execution via dynamic information flow tracking. In *Proceedings of the Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Oct. 2004), 85–96.
20. Vachharajani, N., Bridges, M.J., Chang, J., Rangan, R., Ottoni, G., Blome, J.A., Reis, G.A., Vachharajani, M., 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), 243–254.
21. Vogt, P., Nentwich, F., Jovanovic, N., Kirda, E., Kruegel, C., Vigna, G. Cross-site scripting prevention with dynamic data tainting and static analysis. In *Proceedings of the 14th Network and Distributed System Security Symposium* (2007).
22. Xu, W., Bhatkar, S., Sekar, R. Taint-enhanced policy enforcement: a practical approach to defeat a wide range of attacks. In *Proceedings of the USENIX Security Symposium* (Aug. 2006), 121–136.
23. Yin, H., Song, D., Egele, M., Kruegel, C., Kirda, E. Panorama: capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM Conference on Computer and Communications Security* (2007), 116–127.
24. Yip, A., Wang, X., Zeldovich, N., Kaashoek, M.F. Improving application security with data flow assertions. In *Proceedings of the ACM Symposium on Operating Systems Principles* (Oct. 2009).
25. Zhu, D.Y., Jung, J., Song, D., Kohno, T., Wetherall, D. Tainteraser: protecting sensitive data leaks using application-level taint tracking. *Operating Sys. Rev. 45*, 1 (2011), 142–154.

William Enck (enck@cs.ncsu.edu), Department of Computer Science, North Carolina State University.

Peter Gilbert and Landon P. Cox ([gilbert, lpcox]@cs.duke.edu), Department of Computer Science, Duke University.

Byung-Gon Chun (bgchun@snu.ac.kr), Seoul National University.

Jaeyeon Jung (jjung@microsoft.com), Microsoft Research.

Patrick McDaniel (mcdaniel@cse.psu.edu), Department of Computer Science and Engineering, Pennsylvania State University.

Anmol N. Sheth (anmol.sheth@technicolor.com), Technicolor Research.