TAINTDROID: AN INFORMATION-FLOW TRACKING SYSTEM FOR REAL-TIME PRIVACY MONITORING ON SMARTPHONES

Presented by: Mohsin Junaid
Date: April-17, 2013
TaintDroid

- 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI’10)

- William Enck, Peter Gilbert, Byung-Gon Chun et. al.
Presentation Outline

- Trends in Android
- Motivation and Approach
- taintDroid framework design and challenges
- Experimental setup and findings
- Limitations and related work
Trends in Android

- Number of Android apps in Oct-2012 were 700,000 [news.cnet.com]
Trends in Android

NEW MOBILE THREAT FAMILIES AND VARIANTS RECEIVED PER QUARTER, Q1–Q4 2012

[Graph showing trends in mobile threats by quarter and platform, with Android, Blackberry, iOS, Symbian, and Windows Mobile categories.]
Trends in Android

MOBILE THREATS BY TYPE, 2012

- Trojan 66.1%
- Trojan-Downloader 0.7%
- Trojan-Spy 1.0%
- Adware 2.7%
- Backdoor 0.3%
- Hack-Tool 5.6%
- Monitoring-Tool 7.0%
- Riskware 11.2%
- Spyware 3.7%
- Application 1.7%
Private/sensitive information in Android

- **Device-International Mobile Station Equipment Identity (IMEI)#**
  - Can be used for stolen phones

- **OS Version**
  - Attacker can exploit a bug given the OS version

- **User’s location coordinates**

- **User’s cell phone#**

- **User’s personal information (age, sex, preferences etc.)**
Motivation

- Apps can be installed on a smartphone from GooglePlay, Amazon App store, Mobo-Market etc.

- All of the apps which don’t come with the OS are called third-party apps and hence, are suspicious towards information leakage

- **Monitor** when sensitive data is being leaked in real times from the system through third-party applications
Approach: Dynamic Taint Analysis

- Identify sensitive information which can be used as a source

- Mark these sources as tainted source called taint marking

- Taint other variables as tainted if they are assigned values from tainted source which is called taint propagation

- Dynamic taint analysis tracks/monitors how marked data can reach to any of the sinks which may cause information leakage
Challenges for Monitoring Privacy Info

- **Resource constraints**
  - E.g. tracking Panorama images would be expensive towards performance
  - Battery consumption

- **Third-party apps are entrusted with several types of private information**

- **Sensitive information can be difficult to identify even when it’s sent in clear format**
  - Geo-location data is a pair of floating point numbers

- **Apps can share information**
  - Facebook, twitter, Google search
Issues with existing dynamic taint analysis technique

- Instruction level tracking
  - Too much performance overhead for real time systems

- Taint explosion problem
  - If stack pointer is falsely tainted

- Taint loss problem
  - If complex instruction like CMPXCHG, REP MOV is not properly instrumented
TaintDroid Approach

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

- Android is a Linux-based OS
- All of the core functionality has been written in Java and C/C++
- Applications are written in Java and then are converted into Dalvik Executable (DEX) byte code
- DEX code is executed in Dalvik Virtual Machine (DVM)
- Applications communicate via binder IPC interface
Dalvik VM Interpreter

- DVM is a register based machine while JVM is stack based machine
  - Smartphones are limited in battery resource and register based DVM is faster in executing byte code and hence, saves battery usage

- Each Dex method has its own set of virtual registers

- Registers loosely correspond to local variables of a method

- Interpreter manages method registers via internal execution state stack, so current method registers are always on the stack
Native methods

- Android provides native methods for performance optimization and access to third-party libraries (OpenGL)
- They are written in C/C++
- They expose native functionality to android apps which is provided by the Linux OS
Android background

- Binder IPC

  - Android apps communicate with each other using IPC binder

  - ‘Parcels’ are fundamental components of IPC framework which serialize data objects before sending it out of VM

  - Binder kernel module passes parcel messages between processes
Compiled Language
- After compilation, code is converted into machine language and then stored in an executable file
- Faster in execution but difficult to modify

Interpreted language
- After compilation, code is saved in the same format as written
- It’s easy to modify interpreted code as you don’t have to recompile the code
Android background

- Permissions
  - Some apps don’t request permissions to perform some action, they delegate their job to other apps
  - Facebook didn’t have camera permissions

[applications.androidxiphone.com]
Each app on an android phone is run inside new Dalvik VM sandbox

Each DVM is assigned a unique user id (uid)

All the permissions requested by an app to access phone resources are assigned to uid

Uid remains same when an app is run/updated but pid can be different
Figure 2: TaintDroid architecture within Android.
Framework-implementation challenges

- 1-Taint tag storage

- 2-Taint propagation
  - 2.1-Interpreted code
  - 2.2-Native code
  - 2.3-IPC
  - 2.4- Secondary storage

Figure 2: TaintDroid architecture within Android.
1- Taint Storage

- Typically taint tags are stored for every data byte or word and in non-adjacent memory.

- TaintDroid tags are saved for five different types of data:
  - Method local variables
  - Method arguments
  - Class static fields
  - Class instance fields
  - Arrays
1- Taint Storage

- Local variables and arguments are stored in internal stack of DVM.

- When a method is called, arguments are pushed first and then frame pointer and then local variables.

- So taint tags are stored by doubling the method frame size and then storing tags along with each variable.

- Only one tag is stored for an array for space and performance improvements, though it would increase false positive rate.
1- Taint Storage

- All the addresses which were accessible by fp[i] are now accessible via fp[2.i]

- And DVM is 64-bit machine, so each word is stored in two 32-bit registers, so tainted tag location will be fp[2.i+2]

- Since native methods are not instrumented but tags are patched on return, that’s why their tag storage is a little different

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.
2.1-Interpreted code tag propagation

- TaintDroid primarily tracks primitive data types (int, string etc.) but it also tracks reference objects.

- Local variables and method arguments are denoted by $V_x$

- Class field variables are denoted by $f_x$ (a field variable with class index $x$)

- Class instance fields are denoted by $V_x(f_x)$

- Array variables are denoted by $V_x[.]$
2.1-Interpreted code tag propagation

\[ \tau(v_1) \leftarrow \tau(v_2) \]

indicates that taint tag is being copied from \(v_2\) to \(v_1\)

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.

<table>
<thead>
<tr>
<th>Op Format</th>
<th>Op Semantics</th>
<th>Taint Propagation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>const-op (v_A) C</td>
<td>(v_A \leftarrow C)</td>
<td>(\tau(v_A) \leftarrow \emptyset)</td>
<td>Clear (v_A) taint</td>
</tr>
<tr>
<td>move-op (v_A) (v_B)</td>
<td>(v_A \leftarrow v_B)</td>
<td>(\tau(v_A) \leftarrow \tau(v_B))</td>
<td>Set (v_A) taint to (v_B) taint</td>
</tr>
<tr>
<td>move-op-R (v_A)</td>
<td>(v_A \leftarrow R)</td>
<td>(\tau(v_A) \leftarrow \tau(R))</td>
<td>Set (v_A) taint to return taint</td>
</tr>
<tr>
<td>return-op (v_A)</td>
<td>(R \leftarrow v_A)</td>
<td>(\tau(R) \leftarrow \tau(v_A))</td>
<td>Set return taint ((\emptyset) if void)</td>
</tr>
<tr>
<td>move-op-E (v_A)</td>
<td>(v_A \leftarrow E)</td>
<td>(\tau(v_A) \leftarrow \tau(E))</td>
<td>Set (v_A) taint to exception taint</td>
</tr>
<tr>
<td>throw-op (v_A)</td>
<td>(E \leftarrow v_A)</td>
<td>(\tau(E) \leftarrow \tau(v_A))</td>
<td>Set exception taint</td>
</tr>
<tr>
<td>unary-op (v_A) (v_B)</td>
<td>(v_A \leftarrow \ominus v_B)</td>
<td>(\tau(v_A) \leftarrow \tau(v_B))</td>
<td>Set (v_A) taint to (v_B) taint</td>
</tr>
<tr>
<td>binary-op (v_A) (v_B) (v_C)</td>
<td>(v_A \leftarrow v_B \cup v_C)</td>
<td>(\tau(v_A) \leftarrow \tau(v_B) \cup \tau(v_C))</td>
<td>Update (v_A) taint with (v_B) taint</td>
</tr>
<tr>
<td>binary-op (v_A) (v_B)</td>
<td>(v_A \leftarrow v_A \otimes v_B)</td>
<td>(\tau(v_A) \leftarrow \tau(v_B) \cup \tau(v_A))</td>
<td>Set (v_A) taint to (v_B) taint</td>
</tr>
<tr>
<td>binary-op (v_A) (v_B) (v_C)</td>
<td>(v_A \leftarrow v_B \otimes v_C)</td>
<td>(\tau(v_A) \leftarrow \tau(v_B) \cup \tau(v_C))</td>
<td>Update array (v_B) taint with (v_A) taint</td>
</tr>
<tr>
<td>aput-op (v_A) (v_B) (v_C)</td>
<td>(v_B[v_C] \leftarrow v_A)</td>
<td>(\tau(v_B[\cdot]) \leftarrow \tau(v_B[\cdot]) \cup \tau(v_A))</td>
<td>Set (v_A) taint to array and index taint</td>
</tr>
<tr>
<td>aget-op (v_A) (v_B) (v_C)</td>
<td>(v_A \leftarrow v_B[v_C])</td>
<td>(\tau(v_A) \leftarrow \tau(v_B[v_C]) \cup \tau(v_C))</td>
<td>Set field (f_B) taint to (v_A) taint</td>
</tr>
<tr>
<td>sput-op (v_A) (f_B)</td>
<td>(f_B \leftarrow v_A)</td>
<td>(\tau(f_B) \leftarrow \tau(v_A))</td>
<td>Set field (f_B) taint to (v_A) taint</td>
</tr>
<tr>
<td>sget-op (v_A) (f_B)</td>
<td>(v_A \leftarrow f_B)</td>
<td>(\tau(v_A) \leftarrow \tau(f_B))</td>
<td>Set (v_A) taint to field (f_B) taint</td>
</tr>
<tr>
<td>iput-op (v_A) (v_B) (f_C)</td>
<td>(v_B(f_C) \leftarrow v_A)</td>
<td>(\tau(v_B(f_C)) \leftarrow \tau(v_A))</td>
<td>Set field (f_C) taint to (v_A) taint</td>
</tr>
<tr>
<td>iget-op (v_A) (v_B) (f_C)</td>
<td>(v_A \leftarrow v_B(f_C))</td>
<td>(\tau(v_A) \leftarrow \tau(v_B(f_C)) \cup \tau(v_B))</td>
<td>Set (v_A) taint to field (f_C) and object reference taint</td>
</tr>
</tbody>
</table>
2.1-Interpreted code tag propagation

- Tainting Object References
  - Suppose there is a translation table which converts characters from lower case to upper case characters
  - If a tainted value ‘a’ is used an index to retrieve value from the table, then resulting value ‘A’ will also be considered as tainted
2.1-Interpreted code tag propagation

- Tainting Object References

```java
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); }
    }
}

Object intProxy(int val) { return val; }
int out = (Integer) intProxy(tVal);```
Native code is not monitored the way interpreted code is done in TaintDroid

Return values and external variables are marked as tainted as per the data flow rules given in the table

This tainting is achieved using instrumentation and heuristics depending upon the situational requirements

Two types of native methods:
- Internal VM methods
- JNI methods
2.2-Native code tag propagation

- Internal VM methods

  - Internal VM methods /APIs call native methods directly by passing a pointer to arguments and return values

  - For example, System.arraycopy() native method

  - In Android v2.1, there were 185 native methods, only 5 of them required patching.
2.2-Native code tag propagation

- **JNI methods**
  - JNI methods are invoked through JNI.Call bridge which parses method arguments and assigns return value using method’s descriptor string.
  - JNI call bridge is patched to propagate tags such that whenever method is returned, TaintDroid consults a method profile (from, to) to update taint tags.
  - It assigns union of method argument taint tags to the taint tag of return value.
  - TaintDroid only considers JNI methods which operate on primitive and String arguments/return types.
2.3-IPC tag propagation

- Message level taint granularity to reduce memory and performance overhead

- For example, if one of the variables in the msg is tainted, whole message is marked as tainted

- This increases false positive rate obviously, but that’s trade off between accuracy and performance
2.4 Secondary storage tag propagation

- Taint tag is updated whenever a file is written and tag is propagated on file read

- One tag is reserved for a whole file and stored in extended attributes of the file system.

- To achieve this, extended attribute support for Android’s host file system was implemented

- Coarse-grained granularity: trade off between accuracy and performance
TaintDroid-Framework Design

Figure 2: TaintDroid architecture within Android.
For privacy analysis, taintDroid needs to identify taint sources and sinks and instrument them within OS.

And in Android, private information is obtained either via direct access or service interface, so instrumentation/hook placement would require more careful approach.

Placement w.r.t. information type:
- Low-bandwidth sensors
- High-bandwidth sensors
- Information databases
- Device identifiers
- Network taint sink
Privacy Hook Placement

- **Low-bandwidth sensors**
  - Location and accelerometer information is accessed frequently and via sensor managers, so hook is placed in LocationManager and SensorManager applications.

- **High-bandwidth sensors**
  - Microphone and camera outputs are of higher bandwidth and Android can store them via large data buffers or files or both.
    - So files and data buffers are tainted in these scenarios.
Privacy Hook Placement

- **Information Databases**
  - Address book and SMS messages data is stored in file databases, so the whole file is tainted

- **Device Identifiers**
  - IMEI, IMSI, Phone# are accessed via available APIs, so these APIs are instrumented for adding taint tags

- **Network Taint Sink**
  - TaintDroid identifies privacy leakage when tainted info is transmitted out the network interface
    - So Java framework library is instrumented where native library socket is invoked
Experimental Setup

- 30 apps were selected randomly out of 1100 apps, taken 50 most popular apps from each of 22 categories on Android Market

- Apps were played manually which involved installation, registration if required and exercising the functionality offered by the apps

- Logs were recorded which included tainted binder messages, tainted file output and tainted network messages

- Network traffic using tcpdump was also recorded for verification of results
Findings: Permissions requested by apps

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.

<table>
<thead>
<tr>
<th>Applications*</th>
<th>#</th>
<th>Permissions†</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Weather Channel (News &amp; Weather); Cestos, Solitaire (Game); Movies</td>
<td>6</td>
<td>L C</td>
</tr>
<tr>
<td>(Entertainment); Babble (Social); Manga Browser (Comics)</td>
<td></td>
<td>A P</td>
</tr>
<tr>
<td>Bump, Wertago (Social); Antivirus (Communication); ABC — Animals, Traffic</td>
<td>14</td>
<td>L</td>
</tr>
<tr>
<td>Jam, Hearts, Blackjack. (Games); Horoscope (Lifestyle); Yellow Pages</td>
<td></td>
<td>C A</td>
</tr>
<tr>
<td>(Reference); 3001 Wisdom Quotes Lite, Dastelefonbuch, Astrid (Productivity),</td>
<td></td>
<td>P</td>
</tr>
<tr>
<td>BBC News Live Stream (News &amp; Weather); Ringtones (Entertainment)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Layar (Lifestyle); Knocking (Social); Coupons (Shopping); Trapster (Travel);</td>
<td>6</td>
<td>L</td>
</tr>
<tr>
<td>Spongebob Slide (Game); ProBasketBall (Sports)</td>
<td></td>
<td>C</td>
</tr>
<tr>
<td>MySpace (Social); Barcode Scanner, ixMAT (Shopping)</td>
<td>3</td>
<td>A P</td>
</tr>
<tr>
<td>Evernote (Productivity)</td>
<td>1</td>
<td>L C</td>
</tr>
</tbody>
</table>

* 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
Findings: Information Leakage

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.

<table>
<thead>
<tr>
<th>Observed Behavior (# of apps)</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Phone Information to Content Servers (2)</td>
<td>2 apps sent out the phone number, IMSI, and ICC-ID along with the geo-coordinates to the app’s content server.</td>
</tr>
<tr>
<td>Device ID to Content Servers (7)*</td>
<td>2 Social, 1 Shopping, 1 Reference and three other apps transmitted the IMEI number to the app’s content server.</td>
</tr>
<tr>
<td>Location to Advertisement Servers (15)</td>
<td>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.</td>
</tr>
</tbody>
</table>

* 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).
App data was sent mostly using HTTP Get URLs

Out of 105 connections flagged by taintDroid, 37 were of legitimate use

- These legitimate flags were generated from four apps and OS while using Google Maps for Mobile (GMM) API

- However, taintDroid results showed no false positive as all information leakages were true positive

- It was hard to verify false negatives due to lack of source code availability
Performance

- All experiments were run on Android 2.1 OS, modified for taintDroid

- TaintDroid incurs almost the same performance and memory overhead what an original Android OS does
Performance-MacroBenchmarks

- **Load time:** The duration between when an app is clicked to launch and an activity is displayed.
- **Address Book:** An account creation time (3 SQL queries) and read time (2 SQL queries).
- **Phone Call:** Duration from pressing ‘dial’ button to ‘in-call’ mode.
- **Take Picture:** Duration from pressing ‘Take Picture’ button to re-enabling of ‘Preview’ mode.

**Table 4: Macrobenchmark Results**

<table>
<thead>
<tr>
<th>Test Case</th>
<th>Android</th>
<th>TaintDroid</th>
</tr>
</thead>
<tbody>
<tr>
<td>App Load Time</td>
<td>63 ms</td>
<td>65 ms</td>
</tr>
<tr>
<td>Address Book (create)</td>
<td>348 ms</td>
<td>367 ms</td>
</tr>
<tr>
<td>Address Book (read)</td>
<td>101 ms</td>
<td>119 ms</td>
</tr>
<tr>
<td>Phone Call</td>
<td>96 ms</td>
<td>106 ms</td>
</tr>
<tr>
<td>Take Picture</td>
<td>1718 ms</td>
<td>2216 ms</td>
</tr>
</tbody>
</table>
Performance-Java Microbenchmarks

- **Android port of CaffeineMark 3.0**

- **Sieve:** The classic sieve of eratosthenes finds prime numbers.

- **Loop:** The loop test uses sorting and sequence generation as to measure compiler optimization of loops.

- **Logic:** Tests the speed with which the virtual machine executes decision-making instructions.

- **Method:** The Method test executes recursive function calls to see how well the VM handles method calls.

- **Float:** Simulates a 3D rotation of objects around a point.
Performance-Java Microbenchmarks

- Android port of CaffeineMark 3.0
  - Scores indicate roughly the number of Java instructions executed per second

![Bar chart showing CaffeineMark 3.0 scores for different benchmarks]

Figure 5: Microbenchmark of Java overhead. Error bars indicate 95% confidence intervals.
Performance-IPC Benchmarks

- Client and service applications were developed which perform binder transactions as fast as possible.

- Service manipulates account object (username: String, balance: Integer) by provided interfaces: getAccount() and setAccount().

- Experiment measures the time for client to invoke interface pair 10 times.

<table>
<thead>
<tr>
<th>Table 5: IPC Throughput Test (10,000 msgs).</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time (s)</td>
</tr>
<tr>
<td>----------</td>
</tr>
<tr>
<td>Memory (client)</td>
</tr>
<tr>
<td>Memory (service)</td>
</tr>
</tbody>
</table>
Limitations

- taintDroid tracks only data flows (explicit flows), not control flows.

- Once information leaves the system, it may come back in reply and which may legitimate its usage. TainDroid does not check this kind of leakage.

- TainDroid does not track taint tags on DirectBuffers objects because data is stored in opaque native data structures.

- Mobile Country Code (MCC) and Mobile Network Code (MNC) are typically used for configurations, so during IPC tainting, they produce high false positives.
Limitations/Future research

- All of JNI methods are not enumerated, so remaining methods can be added for more accurate taint tracking.
- Variable level tracking can be done to reduce false positive rate during IPC.
- Applications are played manually which means, this tool is not scalable. So UI testing tool can be used to do the work on a large scale.
- Dynamic analysis cant cover all of the flows, so cfg can be built first and then do dynamic analysis.
Privacy Oracle and TightLip tools can be used for privacy breaching analysis but they can't detect if information is encrypted before being sent.

Haldar et. al instrumented Java String class with taint tracking to prevent SQL Injection attacks.

Language-based information flow security extends existing programming languages by labeling variables with security attributes.

Chandra et. al proposed fine-grained information flow tracking within JVM and instrumented Java Byte code to aid control flow analysis.
Questions