Detecting Android Privacy Leaks via Dynamic Taint Analysis

Jedidiah McClurg, Jonathan Friedman, William Ng

Mentor: Vaibhav Rastogi

Northwestern University
May 28, 2012

Background

- Smartphone shipments increased 42% between 3Q 2010 and 3Q 2011 (Gartner, 11/15/2011)
- Users are increasingly relying on smartphones to manage their personal life
  - Storing and managing contact information of families and friends
  - Reading and keeping up-to-date with personal email on-the-go
  - Using third party application to either provide navigational directions or manage bank accounts

Smartphones store a lot of important information

- For each function, smartphones use and store important personal information of the user. For example:
  - Contact information of friends and families
  - Phone call and text message history
  - Personal emails
  - Browsing history
  - Geographical location
  - Bank account login information
  - Credit card information

Malicious applications can abuse personal information

- Android permissions are vague and complicated to the end user
  - SD card access gives access to everything on the SD card
- It is hard for the user to keep track of how information is being used inside a third-party application
- Malicious applications can either misuse user personal information or leak it to certain destinations

Project Goals

- Keep track of sensitive information and alert the user whenever sensitive information is being leaked from the device (via phone or SMS)
- Create an Android-based system which does not require special permissions or rooting (in contrast to prior research, which requires these features)

Existing work

- Existing research in dealing with this problem:
  - PiOS [Egele et al., NDSS '11]
    - Leaks in iOS applications
  - TaintDroid [Encl et al., OSDI '10]
    - Leaks in Android applications

Our approach

- We focus on Android information leaks
- We implemented a Java application that can insert Dalvik bytecodes into targeted android applications. The leak detection then happens dynamically as the app executes
- It doesn't require root access
- The privacy information are tracked dynamically during runtime of the modified android applications

Compatibility with Android

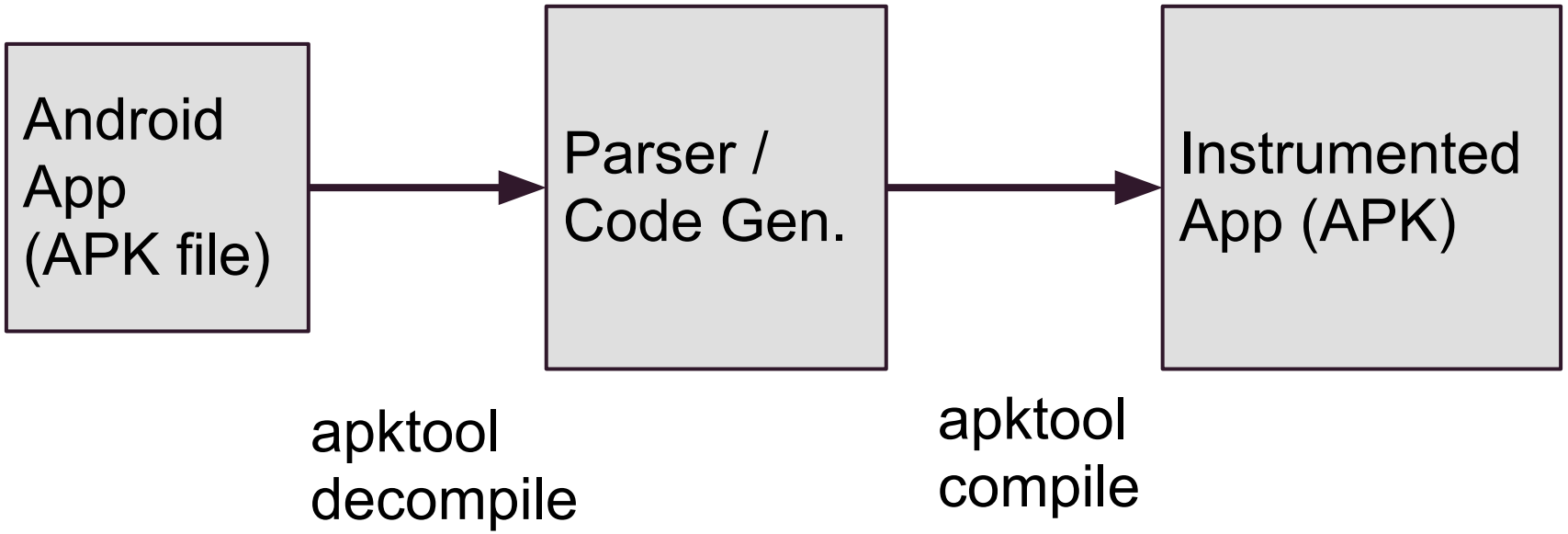
We have created a proof-of-concept App which:

1. Looks for any application package files in two folders: /system/app and /data/app
2. Lists all the contained package files
3. Reads binary data from an APK file
4. *Convert the APK into Smali assembly code and back using baksmali/smali.

*Converting the smali/baksmali JAR files into android dex files is prohibitively slow in Eclipse. Each compilation takes over 30 minutes, which means we are doing development using a PC-based Java app instead. There should be no difference between the two.

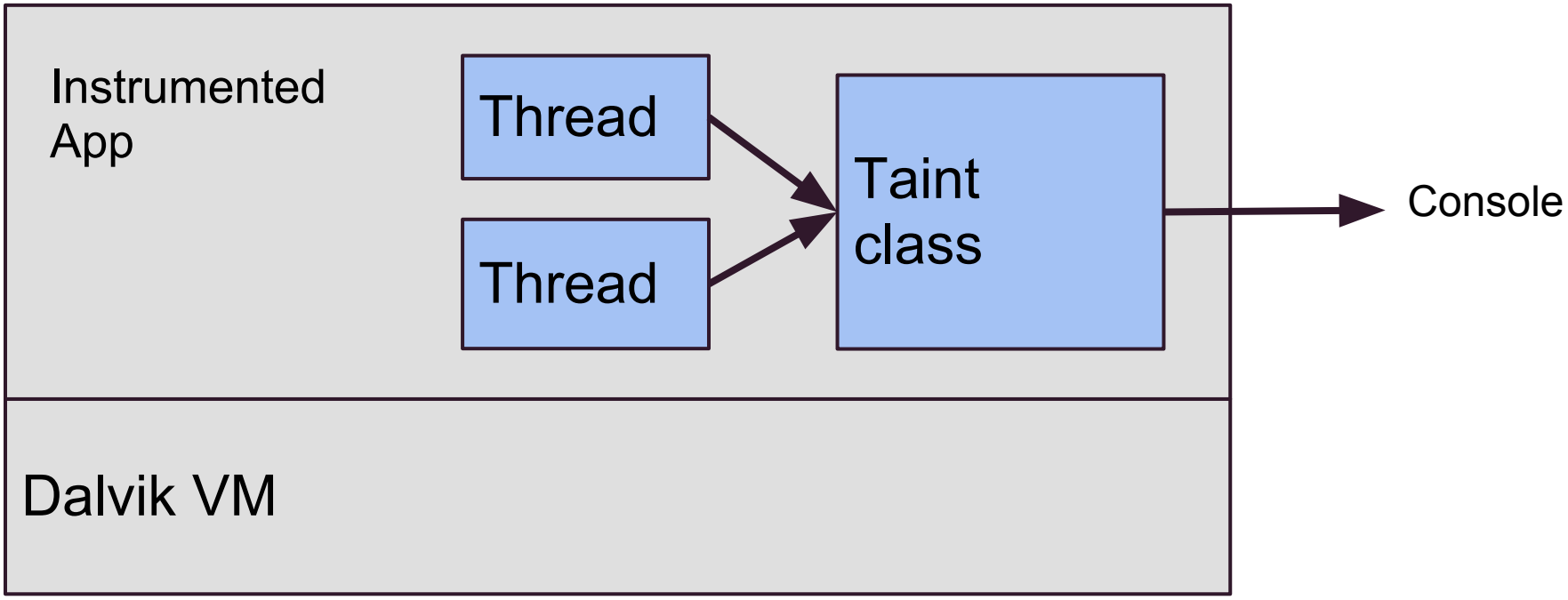
System Overview

- The system produces a new APK which is functionally equivalent to the old APK, but contains its own taint tracking functionality



Running the Analysis

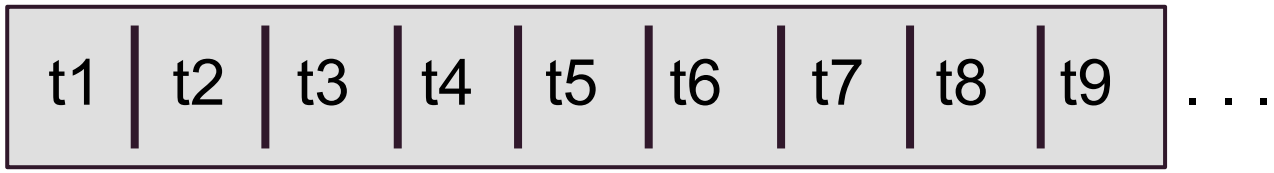
- The instrumented application is executed on the device
- Based on the individual taint semantics for each instruction, the program will communicate taint information with the Taint class
- The Taint class will alert us regarding privacy leaks



Android Decompilation

- The apktool is used to disassemble the APK file into human-readable "smali" files
- One issue is that method parameters are placed *after* local registers
  - To use extra registers, our code instrumentations must increase the local register count
  - But this changes the position of the params, breaking the code
  - Solution: upon method invocation, shift the params to lower-numbered registers

Taint Tracking



- Every object contains a taint value, consisting of a flag for each taint source
- The Taint class contains related functionality:
  - Repository for taint information
  - Specification of taint sources
  - Fine-grained semantics for library methods
- The instrumented application passes references to Taint class to propagate/update their values

Code Instrumentation

- Each instruction is accompanied by a taint value update/propagation
- The instruction sget-object vX, T is followed by a check on T. If T is a taint source, then vX is tainted
- The instruction invoke-virtual {vA, vB, ...}, S->m()X causes vA,vB,... to acquire the union of all their taint values

Live Demo

- Using this basic level of code instrumentation, we have analyzed the Last.fm application
- Defined taint sources (fields)
  - Landroid/os/Build;->MODEL:Ljava/lang/String;
  - Landroid/os/Build$VERSION;->RELEASE:Ljava/lang/String;
  - Lfm/last/android/activity/Profile;->mUsername:Ljava...
- Defined leak rules
  - If "u" is tainted, then u.openConnection() is a privacy leak
  - If "s" tainted, then con.setRequestProperty(s) is a privacy leak

Work to be done before final report

- Finish the taint propagation semantics/instrumentation for each instruction
- Benchmark the instrumented code
- Try to find a way to speed up the Eclipse build so that we can use smali/baksmali from our Android app

Work division among team members

- Jonathan Friedman: created the front-end Android application which will house our taint analysis algorithm and test permissions.
- William Ng: created the Smali code to be inserted into targeted applications to track taint information
- Jedidiah McClurg: wrote the Java code to analyze the applications and instrument the code based on taint propagation semantics and user input

Future Considerations

- Going beyond a proof of concept and creating a fully integrated application that is consumer friendly.
- Fleshing out taint tracking in android system calls
- Creating an alternate version that takes advantage of rooted phones

Conclusion

- We have demonstrated that it is possible to create a system which does taint tracking without special permissions or rooting
- Our system is extensible, and allows for more complex tracking
- We have found information leaks in well-known Android marketplace apps