IJITKM 0973-4414 Volume-8, Number-2 Jan-Jun 2015 pp. 121-130 available online at www.csjournalss.com

Android Application Development and its Security

Krishma
Bhiwani Institute of Technology and Sciences (MDU Rohtak)
krishmatetarwal.k3@gmail.com

Abstract- Android is now the most used mobile operating system in the world. The Google Play app store has been growing at breakneck speed and with almost as many apps as the Apple app store. The fluidity of application markets complicate smart-phone security. Although recent efforts have shed light on particular security issues, there remains little insight into broader security characteristics of smartphone applications. This paper gives a complete knowledge of how to start working on eclipse and develop an application and get it run on emulator. And to better understand smart-phone application security by studying 1,100 popular free Android applications. We introduce the ded decompiler, which recovers Android application source code directly from its installation image. Our analysis uncovered pervasive use/misuse of personal/phone identifiers, and deep penetration of advertising and analytics networks. However, we did not find evidence of malware or exploitable vulnerabilities in the studied applications. We conclude by considering the implications of these preliminary findings and offer directions for future analysis.

I. Introduction

Android is a Linux based, **open source** mobile operating system developed by Open Handset Alliance led by Google to develop apps for Android devices. To start with we use a set of tools that are included in the Android SDK. Once we have downloaded and installed the SDK, we can access these tools right from our Eclipse IDE, through the ADT plugin, or from the command line. Developing with Eclipse is the preferred method because it can directly invoke the tools that we need while developing applications..

The basic steps for developing applications are shown in Figure 1. The development steps encompass four development phases, which include:

Setup: During this phase we install and set up our development environment. We also create Android Virtual Devices (AVDs) and connect hardware devices, on which we can install our applications.

Development: During this phase we set up and develop our Android project, which contains all of the source code and resource files for our application.

Debugging and Testing: During this phase we build our project into a debug gable .apk package that we can install and run on the emulator.

Publishing: During this phase we configure and build our application for release and distribute our application to users.

Application markets such as Apple's App Store and Google's Android Market provide point and click access to hundreds of thousands of paid and free applications.

The fluidity of the markets also presents enormous security challenges. Rapidly developed and deployed applications , coarse permission systems, privacy invading behaviours , malware, and limited security models have led to exploitable phones and applications. Although users seemingly desire it, markets are not in a position to provide security in more than a superficial way. The lack of a common definition for security and the volume of applications ensures that some malicious, questionable, and vulnerable applications will find their way to market.

In this paper, we broadly characterize the security of applications in the Android Market. In this, we make two primary contributions:

- We design and implement a Dalvik decompilier, ded. ded recovers an application's Java source solely from its installation image by inferring lost types, performing DVM-to-JVM bytecode retargeting, and translating class and method structures.

- We analyze 21 million LOC retrieved from the top 1,100 free applications in the Android Market using automated tests and manual inspection. Where possible, we identify root causes and posit the severity of discovered vulnerabilities.

Our popularity focused security analysis provides insight into the most frequently used applications

This paper is an initial but not final word on Android application security. Thus, one should be circumspect about any interpretation of the following results as a definitive statement about how secure applications are today. Rather, we believe these results are indicative of the current state, but there remain many aspects of the applications that warrant deeper analysis.

II. BACKGROUND

Android: Android is an OS designed for smartphones. Depicted in Figure 1, Android provides a sandboxed application execution environment. A customized embedded Linux system interacts with the phone hardware and an off processor cellular radio. The Binder middleware and application API runs on top of Linux. To simplify, an application's only interface to the phone is through these APIs. Each application is executed within a Dalvik Virtual Machine (DVM) running under a unique UNIX uid. The phone comes preinstalled with a selection of *system applications*, e.g., phone dialer, address book.

Applications interact with each other and the phone through different forms of IPC. *Intents* are typed interprocess messages that are directed to particular applications or systems services, or broadcast to applications subscribing to a particular intent type. Persistent *content provider* data stores are queried through SQL-like interfaces. Background *services* provide RPC and callback interfaces that applications use to trigger actions or access data. Finally user interface *activities* receive named *action* signals from the system and other applications.

Binder acts as a mediation point for all IPC. Access to system resources (e.g., GPS receivers, text messaging, phone services, and the Internet), data (e.g., address books, email) and IPC is governed by permissions assigned at install time. The permissions requested by the application and the permissions required to access the application's

interfaces/data are defined in its *manifest* file. To simplify, an application is allowed to access a resource or interface if the required permission allows it.

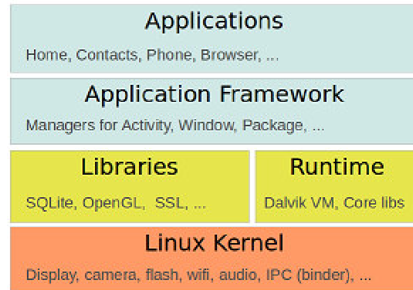


Figure 1: The Android system architecture

Permission assignment—and indirectly the security policy for the phone—is largely delegated to the phone's owner: the user is presented a screen listing the permissions an application requests at install time, which they can accept or reject.

Dalvik Virtual Machine: Android applications are written in Java, but run in the DVM. The DVM and Java byte code runtime environments differ substantially:

Application Structure. Java applications are composed of one or more .class files, one file per class. The JVM loads the bytecode for a Java class from the associated

.class file as it is referenced at run time. Conversely, a Dalvik application consists of a single .dex file containing all application classes.

Figure 2 provides a conceptual view of the compilation process for DVM applications. After the Java compiler creates JVM bytecode, the Dalvik dx compiler consumes the .class files, recompiles them to Dalvik bytecode, and writes the resulting application into a single .dex file. This process consists of the translation, reconstruction, and interpretation of three basic elements of the application: the constant pools, the class definitions, and the data segment. A constant pool describes, not surprisingly, the constants used by a class. This includes, among other items, references to other classes, method names, and numerical constants. The class definitions consist in the basic information such as access flags and class names. The data element contains the method code executed by the target VM, as well as other information related to methods (e.g., number of DVM registers used, local variable table, and operand stack sizes) and to class and instance variables.

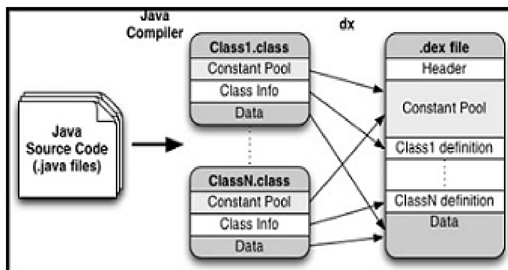


Figure 2: Compilation process for DVM applications

Register architecture The DVM is register-based, whereas existing JVMs are stack-based. Java bytecode can assign local variables to a local variable table before pushing them onto an operand stack for manipulation by opcodes, but it can also just work on the stack without explicitly storing variables in the table. Dalvik bytecode assigns local variables to any of the 2^{16} available registers. The Dalvik opcodes directly manipulate registers, rather than accessing elements on a program stack.

Instruction set The Dalvik bytecode instruction set is substantially different than that of Java. Dalvik has 218 opcodes while Java has 200; however, the nature of the opcodes is very different.Dalvik instructions tend to be longer than Java instructions; they often include the source and destination registers. As a result, Dalvik applications require fewer instructions. In Dalvik bytecode, applications have on average 30% fewer instructions than in Java, but have a 35% larger code size (bytes).

Constant pool structure. Java applications replicate elements in constant pools within the multiple .class files, e.g., referrer and referent method names. The dx compiler eliminates much of this replication. Dalvik uses a single pool that all classes simultaneously reference.

Control flow Structure. Control flow elements such as loops, switch statements and exception handlers are structured differently in Dalvik and Java bytecode. Java bytecode structure loosely mirrors the source code, whereas Dalvik bytecode does not.

Ambiguous primitive types. Java bytecode variable assignments distinguish between integer (int) and single precision floating point (float) constants and between long integer (long) and double precision floating point (double) constants. However, Dalvik assignments (int/float and long/double) use the same opcodes for integers and floats, e.g., the opcodes are untyped beyond specifying precision.

Null references. The Dalvik bytecode does not specify a null type, instead opting to use a zero value constant. Thus, constant zero values present in the Dalvik byte code have ambiguous typing that must be recovered.

Comparison of object references. The Java bytecode uses typed opcodes for the comparison of object references (if acmpeq and if acmpne) and for null comparison of object references (ifnull and ifnonnull). The Dalvik bytecode uses a more simplistic integer comparison for these purposes: a comparison between two integers, and a comparison of an integer and zero respectively. This requires the decompilation process to recover types for integer comparisons used in DVM bytecode.

Storage of primitive types in arrays. The Dalvik byte code uses ambiguous opcodes to store and retrieve elements in arrays of primitive types (e.g., aget for int/float and aget wide for long/double) whereas the corresponding Java bytecode is unambiguous. The array type must be recovered for correct translation.

III. ECLIPSE

Eclipse is an integrated development environment (IDE). It contains a base workspace and an extensible plugin system for customizing the environment. Written mostly in Java, Eclipse can be used to develop applications in Java.

Figure 3. Steps for Application Development

The initial codebase originated from IBM. The Eclipse software development kit (SDK), which includes the Java development tools, is meant for Java developers. Users can extend its abilities by installing plugins written for the Eclipse Platform, such as development toolkits for other programming languages, and can write and contribute their own plugin modules.

Released under the terms of the Eclipse Public License, Eclipse SDK is free and open source software (Table 1).

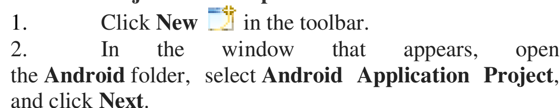
Creating an Android Project
An Android project contains all the files that comprise the source code for Android app. The Android SDK tools make it easy to start a new Android project with a set of default project directories and files.

Eclipse Platform Releases		
Version	**Cryptonym**	**Date of Release**
3.0	-	21-Jun-04
3.1	-	28-Jun-05
3.2	Callisto	30-Jun-06
3.3	Europa	29-Jun-07
3.4	Ganymede	25-Jun-08
3.5	Galileo	24-Jun-09
3.6	Helios	23-Jun-10
3.7	Indigo	22-Jun-11
4.2	Juno	27-Jun-12
4.3	Kepler	26-Jun-13
4.4	Luna	25-Jun-14

Table 1. Eclipse Releases

Create a Project with Eclipse

1. Click **New** in the toolbar.
2. In the window that appears, open the **Android** folder, select **Android Application Project**, and click **Next**.



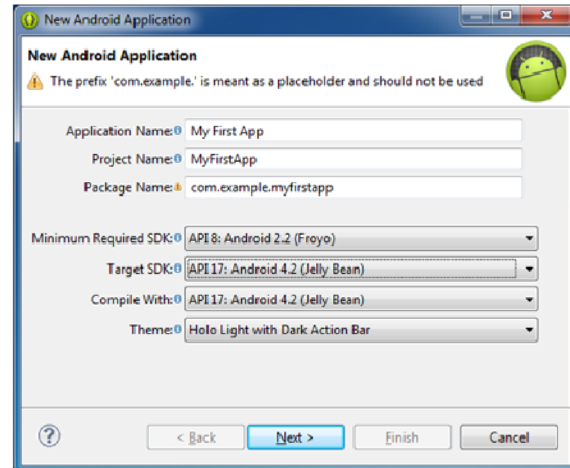


Figure 4. The New Android App Project wizard in Eclipse.

3. Fill in the form that appears:
   - **Application Name** is the app name that appears to users. For this project, use "My First App."
   - **Project Name** is the name of your project directory and the name visible in Eclipse.
   - **Package Name** is the package namespace for your app (following the same rules as packages in the Java programming language). Your package name must be unique across all packages installed on the Android system. For this reason, it's generally best if you use a name that begins with the reverse domain name of your organization or publisher entity. For this project, you can use something like "com.example.myfirstapp." However, you cannot publish your app on Google Play using the "com.example" namespace.
   - **Minimum Required SDK** is the lowest version of Android that your app supports, indicated using the API level. To support as many devices as possible, you should set this to the lowest version available that allows your app to provide its core feature set. If any feature of your app is possible only on newer versions of Android and it's not critical to the app's core feature set, you can enable the feature only when running on the versions that support it. Leave this set to the default value for this project.

- **Target SDK** indicates the highest version of Android (also using the API level) with which you have tested with your application.
As new versions of Android become available, you should test your app on the new version and update this value to match the latest API level in order to take advantage of new platform features.
- **Compile With** is the platform version against which you will compile your app. By default, this is set to the latest version of Android available in your SDK. (It should be Android 4.1 or greater; if you don't have such a version available, you must install one using the SDK Manager). You can still build your app to support older versions, but setting the build target to the latest version allows you to enable new features and optimize your app for a great user experience on the latest devices.
- **Theme** specifies the Android UI style to apply for your app. You can leave this alone.

Click **Next**.

4. On the next screen to configure the project, leave the default selections and click **Next**.

5. The next screen can help you create a launcher icon for your app.

You can customize an icon in several ways and the tool generates an icon for all screen densities. Before you publish your app, you should be sure your icon meets the specifications defined in the Iconography design guide.

Click **Next**.

6. Now you can select an activity template from which to begin building your app.

For this project, select **BlankActivity** and click **Next**.

7. Leave all the details for the activity in their default state and click **Finish**.

Running Your App

How you run your app depends on two things: whether you have a real Android-powered device and whether you're using Eclipse.

Before you run your app, you should be aware of a few directories and files in the Android project:

AndroidManifest.xml

The manifest file describes the fundamental characteristics of the app and defines each of its components

One of the most important elements your manifest should include is the **<uses-sdk>** element. This declares your app's compatibility with different Android versions using the **android:minSdkVersion** and**android:targetSdkVersion** attributes. For our first app, it should look like this:

```
<manifest
xmlns:android="http://schemas.android.com/apk/res/android" ... >
    <uses-sdk android:minSdkVersion="8"
android:targetSdkVersion="17" />
    ...
</manifest>
```

You should always set the **android:targetSdkVersion** as high as possible and test your app on the corresponding platform version.

src/

Directory for your app's main source files. By default, it includes an **Activity** class that runs when your app is launched using the app icon.

res/

Contains several sub-directories for app resources. Here are just a few:

drawable-hdpi/

Directory for drawable objects (such as bitmaps) that are designed for high-density (hdpi) screens. Other drawable directories contain assets designed for other screen densities.

layout/

Directory for files that define your app's user interface.

values/

Directory for other various XML files that contain a collection of resources, such as string and color definitions.

Run on a Real Device

If you have a real Android-powered device, here's how you can install and run your app:

1. Plug in your device to your development machine with a USB cable. If you're developing on Windows, you might need to install the appropriate USB driver for your device.

2. Enable **USB debugging** on your device.

- On most devices running Android 3.2 or older, you can find the option under **Settings > Applications > Development**.
- On Android 4.0 and newer, it's in **Settings > Developer options**.

Note: On Android 4.2 and newer, **Developer options** is hidden by default. To make it available, go to**Settings > About phone** and tap **Build number** seven times. Return to the previous screen to find**Developer options**.

To run the app from Eclipse:

Open one of your project's files and click **Run** from the toolbar.

1. In the **Run as** window that appears, select **Android Application** and click **OK**.

Eclipse installs the app on your connected device and starts it.

Run on the Emulator

To run your app on the emulator you need to first create an Android Virtual Device (AVD). An AVD is a device configuration for the Android emulator that allows you to model different devices.

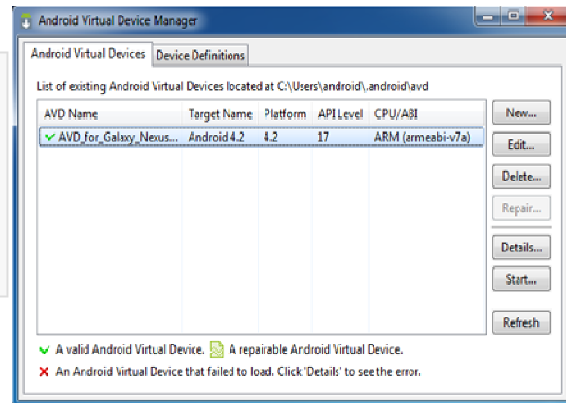


Figure 5 The AVD Manager showing a few virtual devices.

To create an AVD:

1. Launch the Android Virtual Device Manager:

In Eclipse, click Android Virtual Device Manager from the toolbar.

2. In the *Android Virtual Device Manager* panel, click **New**.

3. Fill in the details for the AVD. Give it a name, a platform target, an SD card size, and a skin (HVGA is default).

4. Click **Create AVD**.

5. Select the new AVD from the *Android Virtual Device Manager* and click **Start**.

6. After the emulator boots up, unlock the emulator screen.

To run the app from Eclipse:

1. Open one of your project's files and click **Run** from the toolbar.

2. In the **Run as** window that appears, select **Android Application** and click **OK**.

Eclipse installs the app on your AVD and starts it.

IV. The ded decompiler

Building a decompiler from DEX to Java for the study proved to be surprisingly challenging. Unfortunately, prior to our work, there existed no functional tool for the Dalvik bytecode. Because of the vast differences between JVM and DVM, simple modification of existing decompilers was not possible.

This choice to decompile the Java source rather than operate on the DEX opcodes directly was grounded in two reasons. First, we wanted to leverage existing tools for code analysis. Second, we required access to source code to identify false positives resulting from automated code analysis, e.g., perform manual confirmation.

ded extraction occurs in three stages: *a*) retargeting, *b*) optimization, and *c*) decompilation.

Application Retargeting

The initial stage of decompilation retargets the application .dex file to Java classes. Figure 4 overviews this process: (1) recovering typing information, (2) translating the constant pool, and (3) retargeting the bytecode.

Type Inference: The first step in retargeting is to identify class and method constants and variables. However, the Dalvik bytecode does not always provide enough information to determine the type of a variable or constant from its register declaration. There are two generalized cases where variable types are ambiguous: 1) constant and variable declaration only specifies the variable width (e.g., 32 or 64 bits), but not whether it is a float, integer, or null reference; and 2) comparison operators do not distinguish between integer and object reference comparison (i.e., null reference checks).

Type inference has been widely studied. The seminal Hindley-Milner algorithm provides the basis for type inference algorithms used by many languages such as Haskell and ML. These approaches determine unknown types by observing how variables are used in operations with known type operands. Similar techniques are used by languages with strong type inference, e.g., OCAML, as well weaker inference, e.g., Perl

ded adopts the accepted approach: it infers register types by observing how they are used in subsequent operations with known type operands. Dalvik registers loosely correspond to Java variables. Because Dalvik bytecode reuses registers whose variables are no longer in scope, we must evaluate the register type within its context of the method control flow, i.e., inference must be *path sensitive*

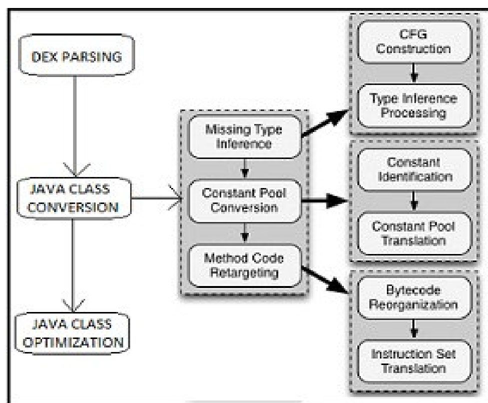


Figure 3: Dalvik bytecode retargeting

There are three ways ded infers a register's type. First, any comparison of a variable or constant with a known type exposes the type. Comparison of dissimilar types requires type coercion in Java, which is propagated to the Dalvik bytecode. Hence legal Dalvik comparisons always involve registers of the same type. Second, instructions such as addint only operate on specific types, manifestly exposing typing information. Third, instructions that pass registers to methods or use a return value expose the type via the method signature.

The ded type inference algorithm proceeds as follows. After reconstructing the control flow graph, ded identifies any ambiguous register declaration. For each such register, ded walks the instructions in the control flow graph starting from its declaration. Each branch of the control flow encountered is pushed onto an inference stack, e.g., ded performs a depth-first search of the control flow graph looking for type exposing instructions. If a type exposing instruction is encountered, the variable is labeled and the process is complete for that variable.

There are three events that cause a branch search to terminate: a) when the register is reassigned to another variable (e.g., a new declaration is encountered), b) when a return function is encountered, and c) when an exception is thrown. After a branch is abandoned, the next branch is popped off the stack and the search continues. Lastly, type information is forward propagated, modulo register reassignment, through the control flow graph from each register declaration to all subsequent ambiguous uses. This algorithm resolves all ambiguous primitive types, except for one isolated case when all paths leading to a type ambiguous instruction originate with ambiguous constant instructions (e.g., all paths leading to an integer comparison originate with registers assigned a constant zero). In this case, the type does not impact decompila-tion, and a default type (e.g., integer) can be assigned.

Constant Pool Conversion: The .dex and .class file constant pools differ in that: *a*) Dalvik maintains a single constant pool for the application and Java maintains one for each class, and *b*) Dalvik bytecode places primitive type constants directly in the bytecode, whereas Java bytecode uses the constant pool for most references. We convert constant pool information in two steps.

The first step is to identify which constants are needed for a .class file. Constants include references to classes, methods, and instance variables. ded traverses the bytecode for each method in a class, noting such references. ded also identifies all constant primitives.

Once ded identifies the constants required by a class, it adds them to the target .class file. For primitive type constants, new entries are created. For class, method, and instance variable references, the created Java constant pool entries are based on the Dalvik constant pool entries. The constant pool formats differ in complexity. Specifically, Dalvik constant pool entries use significantly more references to reduce memory overhead.

Method Code Retargeting: The final stage of the retargeting process is the translation of the method code. First, we preprocess the bytecode to reorganize structures that cannot be directly retargeted. Second, we linearly traverse the DVM bytecode and translate to the JVM.

The preprocessing phase addresses multidimensional arrays. Both Dalvik and Java use blocks of bytecode instructions to create multidimensional arrays; however, the instructions have different semantics and layout. ded reorders and annotates the bytecode with array size and type information for translation.

The bytecode translation linearly processes each Dalvik instruction. First, ded maps each referenced register to a Java local variable table index. Second, ded performs an instruction translation for each encountered Dalvik instruction. As Dalvik bytecode is more compact and takes more arguments, one Dalvik instruction frequently expands to multiple Java instructions. Third, ded patches the relative offsets used for branches based on preprocessing annotations. Finally, ded defines exception tables that describe try/catch/finally blocks. The resulting translated code is combined with the constant pool to creates a legal Java .class file.

Optimization and Decompilation

At this stage, the retargeted .class files can be decompiled using existing tools, e.g., Fernflower or Soot . However, ded's bytecode translation process yields unoptimized Java code. For example, Java tools often optimize out unnecessary assignments to the local variable table, e.g., unneeded return values. Without optimization, decompiled code is complex and frustrates analysis. Furthermore, artifacts of the retargeting pro-cess can lead to decompilation errors in some decompilers. The need for bytecode optimization is easily demonstrated by considering decompiled loops. Most decompilers convert for loops into infinite loops with break instructions. While the resulting source code is functionally equivalent to the original, it is significantly more difficult to understand and analyze, especially for nested loops. Thus, we use Soot as a post-retargeting optimizer. While Soot is centrally an optimization tool with the ability to recover source code in most cases, it does not process certain legal program idioms (bytecode structures) generated by ded. In particular, we encountered two central problems involving, 1) interactions between synchronized blocks and exception handling, and 2) complex control flows caused by break statements. While the Java bytecode generated by ded is legal, the source code failure rate reported in the following section is almost entirely due to Soot's inability to extract source code from these two cases. We will consider other decompilers in future work, e.g., Jad , JD , and Fernflower .

V. Source Code Recovery Validation

We have performed extensive validation testing of ded . The included tests recovered the source code for small, medium and large open source applications and found no errors in recovery. In most cases the recovered code was virtually indistinguishable from the original source (modulo comments and method local variable names, which are not included in the bytecode).

Retargeting Failures. 0.59% of classes were not retar-geted. These errors fall into three classes: *a*) unresolved references which prevent optimization by Soot, *b*) type violations caused by Android's dex compiler and *c*) ex-tremely rare cases in which ded produces illegal byte-code. Recent efforts have focused on improving opti-mization, as well as redesigning ded with a formally de-fined type inference apparatus. Parallel work on improv-ing ded has been able to reduce these errors by a third, and we expect further improvements in the near future.

Decompilation Failures. 5% of the classes were suc-cessfully retargeted, but Soot failed to recover the sourcecode. Here we are limited by the state of the art in de-compilation. In order to understand the impact of de-compiling ded retargeted classes verses ordinary Java .class files, we performed a parallel study to evaluate Soot on Java applications generated with traditional Java compilers. Of 31,553 classes from a variety of packages, Soot was able to decompile 94.59%, indicating we cannot do better while using Soot for decompilation.

VI. Evaluating Android Security

Our Android application study consisted of a broad range of tests focused on three kinds of analysis: *a*) exploring issues uncovered in previous studies and malware advisories, *b*) searching for general coding security failures, and *c*) exploring misuse/security failures in the use of Android framework. The following discusses the process of identifying and encoding the tests.

i. Analysis Specification

We used four approaches to evaluate recovered source code: *control flow analysis*, *data flow analysis*, *structural analysis*, and *semantic analysis*. Unless otherwise specified, all tests used the Fortify SCA static analysis suite, which provides these four types of analysis. The following discusses the general application of these approaches. The details for our analysis specifications can be found in the technical report

Control flow analysis. Control flow analysis imposes constraints on the sequences of actions executed by an input program *P*, classifying some of them as errors. Essentially, a control flow rule is an automaton *A* whose input words are sequences of actions of *P*—i.e., the rule *monitors* executions of *P*. An erroneous action sequence is one that drives *A* into a predefined *error state*. To statically detect violations specified by *A*, the program analysis traces each control flow path in the tool's model of *P*, synchronously "executing" *A* on the actions executed along this path. Since not all control flow paths in the model are feasible in concrete executions of *P*, false positives are possible. False negatives are also possible in principle, though uncommon in practice. Figure 4 shows an example automaton for sending intents. Here, the error state is reached if the intent contains data and is sent unprotected without specifying the target component, resulting in a potential unintended information leakage.

Data flow analysis. Data flow analysis permits the declarative specification of problematic data flows in the input program. For example, an Android phone contains several pieces of private information that should never leave the phone: the user's phone number, IMEI (device ID), IMSI (subscriber ID), and ICC-ID (SIM card serial number). In our study, we wanted to check that this information is not leaked to the network. While this property can in principle be coded using automata, data flow specification allows for a much easier encoding. The specification declaratively labels program statements matching certain syntactic patterns as *data flow sources* and *sinks*. Data flows between the sources and sinks are violations.

Structural analysis. Structural analysis allows for declarative pattern matching on the abstract syntax of the input source code. Structural analysis specifications are not concerned with program executions or data flow, therefore, analysis is local and straightforward. For example, in our study, we wanted to specify a bug pattern where an Android application mines the device ID of the phone on which it

runs. This pattern was defined using a structural rule that stated that the input program called a method *getDeviceId()* whose enclosing class was *android.telephony.TelephonyManager*.

Semantic analysis. Semantic analysis allows the specification of a limited set of constraints on the values used by the input program. For example, a property of interest in our study was that an Android application does not send SMS messages to hard-coded targets. To express this property, we defined a pattern matching calls to Android messaging methods such as *sendTextMessage()*. Seman-tic specifications permit us to directly specify that the first parameter in these calls (the phone number) is not a constant. The analyzer detects violations to this property using constant propagation techniques well known in program analysis literature.

ii. Analysis Overview

Our analysis covers both dangerous functionality and vulnerabilities. Selecting the properties for study was a significant challenge. For brevity, we only provide an overview of the specifications. The technical report provides a detailed discussion of specifications.

Misuse of Phone Identifiers Previous studies identified phone identifiers leaking to remote network servers. We seek to identify not only the existence of data flows, but understand why they occur.

Exposure of Physical Location Previous studies identified location exposure to advertisement servers. Many applications provide valuable location-aware utility, which may be desired by the user. By manually inspecting code, we seek to identify the portion of the application responsible for the exposure.

Abuse of Telephony Services Smart-phone malware has sent SMS messages to premium-rate numbers. We study the use of hard-coded phone numbers to identify SMS and voice call abuse.

Eavesdropping on Audio/Video Audio and video eavesdropping is a commonly discussed smart-phone threat. We examine cases where applications record audio or video without control flows to UI code.

Botnet Characteristics (Sockets) PC botnet clients historically use non HTTP ports and protocols for command and control. Most applications use HTTP client wrappers for network connections, therefore, we examine *Socket* use for suspicious behavior.

Harvesting Installed Applications The list of installed applications is a valuable demographic for marketing. We survey the use of APIs to retrieve this list to identify harvesting of installed applications.

Use of Advertisement Libraries Previous studies identified information exposure to ad and analytics networks. We survey inclusion of ad and analytics libraries and the information they access.

Dangerous Developer Libraries During our manual source code inspection, we observed dangerous functionality replicated between applications. We re-port on this replication and the implications.

Android-specific Vulnerabilities We search for non-secure coding practices, including: writing sensitive information to logs, unprotected broadcasts of information, IPC null checks, injection at-tacks on intent actions, and delegation.

General Java Application Vulnerabilities. We look for general Java application vulnerabilities, including misuse of passwords, misuse of cryptography, and traditional injection vulnerabilities. Due to space limitations, individual results for the general vulnerability analysis are reported in the technical report.

VII. Application Analysis Results

In this section, we document the program analysis results and manual inspection of identified violations.

Table 2: Access of Phone Identifier APIs

Identifier	# Calls	# Apps	# w/ Permission
Phone Number	167	129	105
IMEI	378	216	184†
IMSI	38	30	27
ICC-ID	33	21	21
Total Unique	-	246	210†

Defined as having the READ_PHONE_STATE permission.

† Only 1 app did not also have the INTERNET permission.

i. Information Misuse

In this section, we explore how sensitive information is being leaked through information sinks including *OutputStream* objects retrieved from *URLConnection*s, HTTP GET and POST parameters in *HttpClient* connections, and the string used for *URL* objects. Future work may also include SMS as a sink.

ii. Phone Identifiers

We studied four phone identifiers: phone number, IMEI (device identifier), IMSI (subscriber identifier), and ICC-ID (SIM card serial number). We performed two types of analysis: *a*) we scanned for APIs that access identifiers, and *b*) we used data flow analysis to identify code capable of sending the identifiers to the network.

Table 2 summarizes APIs calls that receive phone identifiers. In total, 246 applications (22.4%) included code to obtain a phone identifier; however, only 210 of these applications have the READ_PHONE_STATE permission required to obtain access. Section 5.3 discusses code that probes for permissions. We observe from Table 2 that applications most frequently access the IMEI (216 applications, 19.6%). The phone number is used second most (129 applications, 11.7%). Finally, the IMSI and ICC-ID are very rarely used (less than 3%).

Table 3 indicates the data flows that exfiltrate phone identifiers. The 33 applications have the INTERNET permission, but 1 application does not have the READ_ PHONE_STATE permission. We found data flows for all four identifier types: 25 applications have IMEI data flows; 10 applications have phone number data flows; 5 applications have IMSI data flows; and 4 applications have ICC-ID data flows.

To gain a better understanding of how phone identifiers are used, we manually inspected all 33 identified applications, as well as several additional applications that contain calls to identifier APIs. We confirmed exfiltration for all but one application. In this case, code complexity hindered manual confirmation; however we identified a different data flow not found by program analysis. The analysis informs the following findings.

Finding 1 - *Phone identifiers are frequently leaked through plaintext requests.* Most sinks are HTTP GET or POST parameters. HTTP parameter names

Table 3: Detected Data Flows to Network Sinks

Sink	Phone Identifiers		Location Info.	
	# Flows	# Apps	# Flows	# Apps
OutputStream	10	9	0	0
HttpClient Param	24	9	12	4
URL Object	59	19	49	10
Total Unique	-	33	-	13

for the IMEI include: "uid," "user-id," "imei," "deviceId," "deviceSerialNumber," "devicePrint," "X-DSN," and "uniquely code"; phone number names include "phone" and "mdn"; and IMSI names include "did" and "imsi." In one case we identified an HTTP parameter for the ICC-ID, but the developer mislabeled it "imei."

Finding 2 - *Phone identifiers are used as device fingerprints.* Several data flows directed us towards code that reports not only phone identifiers, but also other phone properties to a remote server. For example, a wall-paper application (com.eoeandroid.eWallpapers.cartoon) contains a class named *SyncDeviceInfosService* that collects the IMEI and attributes such as the OS ver-sion and device hardware. The method *sendDevice-Infos()* sends this information to a server. In an-other application (com.avantar.wny), the method *PhoneStats.toUrlFormatedString()* creates a URL parameter string containing the IMEI, device model, platform, and application name. While the intent is not clear, such fingerprinting indicates that phone identifiers are used for more than a unique identifier.

Finding 3 - *Phone identifiers, specifically the IMEI, are used to track individual users.* Several applications contain code that binds the IMEI as a unique identifier to network requests. For example, some applications (e.g. com.Qunar and com.nextmobileweb.craigsphone) appear to bundle the IMEI in search queries; in a travel application (com.visualit.tubeLondonCity), the method *refreshLive-Info()* includes the IMEI in a URL; and a "keyring" application (com.froogloid.kring.google.zxing.client.android) appends the IMEI to a variable named *retailer-LookupCmd.* We also found functionality that in-cludes the IMEI when checking for updates (e.g., com.webascender.callerid, which also includes the phone number) and retrieving advertisements (see Find-ing 6). Furthermore, we found two applications (com.taobo.tao and raker.duobao.store) with network ac-cess wrapper methods that include the IMEI for all con-nections. These behaviors indicate that the IMEI is used as a form of "tracking cookie".

Finding 4 - *The IMEI is tied to personally identifi-able information (PII).* The common belief that the IMEI to phone owner mapping is not visible outside the cellular network is no longer true. In several cases, we found code that bound the IMEI to account information and other PII. For example, applications (e.g. com.slacker.radio and com.statefarm.pocketagent) include the IMEI in account registration and login re-quests. In another application (com.amazon.mp3), the method *linkDevice()* includes the IMEI. Code inspec-tion indicated that this method is called when the user chooses to "Enter a claim code" to redeem gift cards. We also found IMEI use in code for sending comments and reporting problems (e.g., com.morbe.guarder and com.fm207.discount). Finally, we found one application (com.andoop.highscore) that appears to bundle the IMEI when submitting high scores for games. Thus, it seems clear that databases containing mappings between physical users and IMEIs are being created.

Finding 5 - *Not all phone identifier use leads to exfiltration.* Several applications that access phone identifiers did not exfiltrate the values. For example, one application (com.amazon.kindle) creates a device fingerprint for a verification check. The fingerprint is kept in "secure storage" and does not appear to leave the phone. An-other application (com.match.android.matchmobile) as signs the phone number to a text field used for account registration. While the value is sent to the network during registration, the user can easily change or remove it.

Finding 6 - *Phone identifiers are sent to advertisement and analytics servers.* Many applications have custom ad and analytics functionality. For example, in one application (com.accuweather.android), the class *ACCUWX AdRequest* is an IMEI data flow sink. Another application (com.amazon.mp3) defines Android service component *AndroidMetricsManager*, which is an IMEI data flow sink. Phone identifier data flows also occur in ad libraries. For example, we found a phone num-

ber data flow sink in the com/wooboo/adlib_android library used by several applications (e.g., cn.ecook, com.superdroid.sqd, and com.superdroid.ewc). Section 5.3 discusses ad libraries in more detail.

Location Information

Location information is accessed in two ways: (1) calling *getLastKnownLocation()*, and (2) defining callbacks in a *LocationListener* object passed to *requestLocationUpdates()*. Due to code recovery failures, not all *Location-Listener* objects have corresponding *requestLocationUpdates()* calls. We scanned for all three constructs.

Table 4 summarizes the access of location information. In total, 505 applications (45.9%) attempt to access location, only 304 (27.6%) have the permission to do so. This difference is likely due to libraries that probe for permissions, as discussed in Section 5.3. The separation between *LocationListener* and *requestLocationUpdates()* is primarily due to the AdMob library, which de-fined the former but has no calls to the latter.

To overcome missing code challenges, the data flow source was defined as the *getLatitude()* and *getLongitude()* methods of the *Location* object retrieved from the location APIs. We manually inspected the 13 applications with location data flows. Many data flows appeared to reflect legitimate uses of location for weather, classifieds, points of interest, and social networking services. Inspection of the remaining applications informs the following findings:

Finding 7 - *The granularity of location reporting may not always be obvious to the user.* In one application (com.andoop.highscore) both the city/country *and* geographic coordinates are sent along with high scores. Users may be aware of regional geographic information associated with scores, but it was unclear if users are aware that precise coordinates are also used.
Finding 8 - *Location information is sent to advertisement servers.* Several location data flows appeared to terminate in network connections used to retrieve ads. For example, two applications (com.avantar.wny and com.avantar.yp) appended the location to the variable *webAdURLString*. Motivated by, we inspected the AdMob library to determine why no data flow was found and determined that source code recovery failures led to the false negatives. Section 5.3 expands on ad libraries.

Phone Misuse
This section explores misuse of the smartphone interfaces, including telephony services, background recording of audio and video, sockets, and accessing the list of installed applications.

VIII. Study Limitations

Our study section was limited in three ways: *a*) the studied applications were selected with a bias towards popularity; *b*) the program analysis tool cannot compute data and control flows for IPC between components; and *c*) source code recovery failures interrupt data and control flows. Missing data and control flows may lead to false negatives. In addition to the recovery failures, the program analysis tool could not parse 8,042 classes, reducing coverage to 91.34% of the classes.
Additionally, a portion of the recovered source code was obfuscated before distribution. Code obfuscation significantly impedes manual inspection. It likely exists to protect intellectual property; Google suggests obfuscation using ProGuard (proguard.sf.net) for applications using its licensing service. ProGuard protects against readability and does not obfuscate control flow. Therefore it has limited impact on program analysis.
Many forms of obfuscated code are easily recognizable: e.g., class, method, and field names are converted to single letters, producing single letter Java filenames (e.g., a.java). For a rough estimate on the use of obfuscation, we searched applications containing a.java. In total, 396 of the 1,100 applications contain this file. As discussed in Section 5.3, several advertisement and analytics libraries are obfuscated. To obtain a closer estimate of the number of applications whose main code is obfuscated, we searched for a.java within a file path equivalent to the package name (e.g., com/foo/appname for com.foo.appname). Only 20 applications (1.8%) have this obfuscation property, which is expected for free applications (as opposed to paid applications). However, we stress that the a.java heuristic is not intended to be a firm characterization of the percentage of obfuscated code, but rather a means of acquiring insight.

IX. What This All Means

Identifying a singular take away from a broad study such as this is non obvious. We come away from the study with two central thoughts; one having to do with the study apparatus, and the other regarding the applications. ded and the program analysis specifications are enabling technologies that open a new door for application certification. We found the approach rather effective despite existing limitations. In addition to further studies of this kind, we see the potential to integrate these tools into an application certification process. We leave such discussions for future work, noting that such integration is challenging for both logistical and technical reasons .On a technical level, we found the security characteristics of the top 1,100 free popular applications to be consistent with smaller studies. Our findings indicate an overwhelming concern for misuse of privacy sensitive information such as phone identifiers and location information. One might speculate this occur due to the difficulty in assigning malicious intent.
Arguably more important than identifying the existence the information misuse, our manual source code inspection sheds more light on *how* information is misused. We found phone identifiers, e.g., phone number, IMEI, IMSI, and ICC-ID, were used for everything from “cookieesque” tracking to account numbers. Our findings also support the existence of databases external to cellular providers that link identifiers such as the IMEI to personally identifiable information.
Our analysis also identified significant penetration of ad and analytic libraries, occurring in 51% of the studied applications. While this might not be surprising for free applications, the number of ad and analytics libraries included per application was unexpected. One application included as many as eight different libraries. It is unclear why an application needs more than one advertisement and one analytics library.
From a vulnerability perspective, we found that many developers fail to take necessary security precautions. For example, sensitive information is frequently written to Android’s centralized logs, as well as occasionally broadcast to unprotected IPC. We also identified the potential for IPC injection attacks; however, no cases were readily exploitable.
Finally, our study only characterized one edge of the application space. While we found no evidence of telephony misuse, background recording of audio or video, or abusive network connections, one might argue that such malicious functionality is less likely to occur in popular applications. We focused our study on popular applications to characterize those most frequently used. Future studies should take samples that span application popularity. However, even these samples may miss the existence of truly malicious applications. Future studies should also consider several additional attacks, including installing new applications , JNI execution , address book exfiltration, destruction of SDcard contents, and phishing .

X. Related Work

Many tools and techniques have been designed to identify security concerns in software. Software written in C is particularly susceptible to programming errors that result in vulnerabilities. Ashcraft and Engler use compiler extensions to identify errors in range checks. MOPS uses model checking to scale to large amounts of source code . Java applications are inherently safer than C applications and avoid simple vulnerabilities such as buffer overflows. Ware and Fox compare eight different open source and commercially available Java source code analysis tools, finding that no one tool detects all vulnerabilities. Hovemeyer and Pugh study six popular Java applications and libraries using FindBugs extended with additional checks. While analysis included non security bugs, the results motivate a strong need for automated analysis by all

developers. Livshits and Lam focus on Java based Web applications. In the Web server environment, inputs are easily controlled by an adversary, and left unchecked can lead to SQL injection, crosssite scripting, HTTP response splitting, path traversal, and command injection. Felmetsger et al also study Java based web applications; they advance vulnerability analysis by providing automatic detection of application specific logic errors.

Spyware and privacy breaching software have also been studied. Kirda et al. consider behavioral properties of BHOs and toolbars. Egele et al. target information leaks by browser based spyware explicitly using dynamic taint analysis. Panaorama considers privacy breaching malware in general using whole system, fine grained taint tracking. Privacy Oracle uses differential black box fuzz testing to find privacy leaks in applications.

On smartphones, TaintDroid uses system wide dynamic taint tracking to identify privacy leaks in Android applications. By using static analysis, we were able to study a far greater number of applications (1,100 vs. 30). However, TaintDroid's analysis confirms the exfiltration of information, while our static analysis only confirms the potential for it. Kirin also uses static analysis, but focuses on permissions and other application configuration data, whereas our study analyzes source code. Finally, PiOS performs static analysis on iOS applications for the iPhone. The PiOS study found the majority of analyzed applications to leak the device ID and over half of the applications include advertisement and analytics libraries.

XI. Conclusions

objective behind this paper presentation was to discuss all basic details to start android application and to overcome the technical jargons which come as a big constraint on the way of beginner programmer. Smartphones are rapidly becoming a dominant computing platform. Low barriers of entry for application developers increases the security risk for end users. In this paper, we described the ded decompiler for Android applications and used decompiled source code to perform a breadth study of both dangerous functionality and vulnerabilities. While our findings of exposure of phone identifiers and location are consistent with previous studies, our analysis framework allows us to observe not only the existence of dangerous functionality, but also how it occurs within the context of the application.

Moving forward, we foresee ded and our analysis specifications as enabling technologies that will open new doors for application certification. However, the integration of these technologies into an application certification process requires overcoming logistical and technical challenges. Our future work will consider these challenges, and broaden our analysis to new areas, including application installation, malicious JNI, and phishing.

XII. References

[1] http://developer.android.com

[2]nternational Journal of Scientific and Research Publications,Volume 4, Issue 2, February 2014,ISSN 2250-3153

[3] STORM, D. Zombies and Angry Birds attack: mobile Phone malware. Computerworld (November 2010).