A quick guide to Android app reversing

What is byte code

Android bytecode is based on Java. It is a virtual machine that has assembly-like syntax. Registers can store primitives or objects, so registers have a type. Errors occur if registers are used such that a type error would occur. This is checked at run-time.
Methods can be called on instances and arguments can be passed to them. The syntax is that of C-style function invocations, meaning that the "this" pointer is the first argument, followed by the rest of the arguments. Static functions don't pass a "this".

Viewing the Manifest

The file `AndroidManifest.xml` is a description of the app. It has alot of information such as permissions requested, all the content providers and receivers, and other things that need to be statically declared. This is an XML file but appears in binary in the APK. To view the actual data:
`aapt d xmltree filename.apk AndroidManifest.xml`

Decompiling the APK

APKs are zip file archives that store the android app. They are no longer the java source code files, so decompiling them only gives you the "compiled" byte code. The tool `apktool` supports decompiling:
`$ apktool file.apk`
Use `-r` to avoid decompiling the resources (e.g., images, etc.). This is useful if you want to later re-compile it because you made changes, etc.

Directory Structure

The decompiled apk will have a bunch of directories. The ones containing code are in `smali/` and `smali_classesX` for some number X. These contain the directory structure matching java's classpath. So if there is code from the ad library flurry with java package `com.flurry.ClassName`, you will see something like `smali/com/flurry/ClassName.smali`

Hello World

```
# comments
.class public LHelloWorld;
# classes are formatted as Lclassname;
# e.g., Ljava/lang/String;
# This is how it is used in reflection.

.super Ljava/lang/Object;
# i.e., class HelloWorld extends Object

.method public static main([Ljava/lang/String;)V
# main function takes array of Strings, returns void
.locals 2  # two stack registers for all variables
# these are v0 and v1.

sget-object v0, Ljava/lang/System;->out:Ljava/io/PrintStream;
# put in register v0 the object System.out
# it has type java.io.PrintStream
# the colon says out is a member variable of System

const-string v1, "Hello World!"
# v1 is now that string

invoke-virtual {v0, v1},
      Ljava/io/PrintStream;->println(Ljava/lang/String;)V
# function call! Call println's like this:
#    v0.println(v1)
# v1 is a String and it returns void and always
# are specified

return-void
# end of the function
.end method
```

Smali Syntax

The registers are indexed from v0 onwards. The number of them is equal to the value of `.locals`. In addition, there are registers p0, p1, etc., for all the parameters to the function. The class instance (this) gets p0, then p1 is the first argument. If the function is static, then p0 becomes the first parameter instead.

The official bytecode reference is here:

`https://source.android.com/devices/tech/dalvik/dalvik-bytecode.html`

another site with reference:

`http://pallergabor.uw.hu/androidblog/dalvik_opcodes.html`

Some gotchas:

- When used for primitive types (e.g., ints) registers are 32-bits wide. This means that if you store a 64-bit primitive (long and doubles) it actually uses two adjacent registers. This throws off your counting if you don't account for that.
- Labels like `:cond_0` are places that can be jumped to. Lines like `if-le v1, p2, :cond_0` mean that if v1 is less than or equal to p2, then go to cond_0, otherwise go to the next line.
- To create a new class, you use the line `new-instance v2, Ljava/lang/StringBuilder;`. This doesn't call the constructor, however, which is then done with `invoke-direct v2, Ljava/lang/StringBuilder;-><init>()V`.
- If a function returns a value, you need to specify where to put it if you want to use it. After an invoke line, use `move-result v0` to have v0 now store the result of the previous function call.
- If you see that some variable is used, search for wherever `iput-object` is used to set it, as well as whenever `iget-object` is used to get it and them call some methods on it to modify it.
- Unlike in C code, the use of XOR in java is super rare. If you search for it you can often find things like roll-your-own crypto or other obfuscation methods.
- Sometimes encryption keys and such things are encoded as an array of bytes. Java uses signed 8-bit values in two's-complement, so you need to add 256 to a negative value. I wrote a tool to convert these from .smali files to a hexstring (https://github.com/clambassador/verkfaery/)

code	value
D	double
J	long
V	void (only for return value)
I	int
Z	bool
Lclassname;	instance of classname
B	byte
S	short
C	char
F	float

Obfuscation

You may quickly find that decompiled code looks like this:

```
.line 494
iget-object v0, p0, Lm/f/jt;->a:Lm/f/jp;
invoke-static {v0}, Lm/f/jp;->f(Lm/f/jp;)Lm/f/ck;
move-result-object v0
iget-object v1, p0, Lm/f/jt;->a:Lm/f/jp;
iget-object v1, v1, Lm/f/jp;->c:Lcom/metafun/fun/ads/model/AdData;
invoke-virtual {v0, v1}, Lm/f/ck;->onAdClicked(Lcom/metafun/fun/ads/model/AdBase;)V
```

This first line says that the class `m.f.jt` has a member variable `a` of type `m.f.jp`. What? The reason is that many apps will intentionally obfuscate their code by removing all the friendly names corresponding to the variables, methods, and classes. This makes figuring out the purpose of the code much harder. This is the reverse engineering art part. It takes time to explore, figure out what each part does. Write things down when you figure it out. Sometimes these are just front ends for API calls, and public API calls cannot be obfuscated. Look for string constants that are embedded within.

Changing the APK

You can add your own smali code into the program. So if you wanted to recompile an app with additional logging statements, you can. For example, suppose there was a register v2 of type java.lang.String, and you wanted to log it using the Android logging system. You could add:

```
const-string v1, "TEST"
invoke-static {v1, v2}, Landroid/util/Log;->i(Ljava/lang/String;Ljava/lang/String;)I
```

to the code to get it to call the logging line here. Keep in mind that whenever you set a value to a register, like setting v1 to TEST, it overwrites what was there before. This can really screw up programs, so you need to make sure that the value isn't needed. You can either use a register that is about to be overwritten in a few lines, or you can increase the number of registers. There is an upper bound on the number of registers, and this includes the parameters, so this approach does not always work.

Recompiling the APK

Apks have to be signed by the developer. But you can be your own developer. But you still have to sign them. So after you have decompile the apk, you can recompile it as follows: `apktool b <path-to-decompiled> -o unsigned.apk`
A signing program is available from the following git repository:

`https://github.com/glitterballs/release-tools.git`

Go to its SignApk directory and run the following:

`java -jar signapk.jar certificate.pem key.pk8 unsigned.apk signed.apk`

You can create your own certificate and key to do the signing if you want.

Running the APK

The `adb` command line tool is responsible for all android to computer communication. It supports a great number of commands. To install an app, use `adb install -r filename.apk` (e.g., the signed.apk from release-tools). The -r flag replaces it if its already there. To view the logfile, `adb logcat`. You can also call `adb shell` to spawn a shell, and from there you effectively have a linux terminal environment.

Searching Code

Grep is your friend: `grep -r something` will search all files in the directory and subdirectory (r=recursive) for the string something. Doing `grep -ri` makes it case *i*nsensitive. This points you to the files where, for example, a string literal is loaded into code.

To see all the strings grep for the *const-string* opcode.

Hidden Logging

Many apps have alot of logging statements commented out. If you ever notice in the code that you see generic-looking tag-like strings being passed as argument one alongside a more specific information message as argument two, this can be an apps on logging framework. Normally logging is `Log.i(TAG, message)`; some apps will have their own wrapper for that so that they can easily "comment" out all logging statements simply by setting a boolean to not actually call the Log.i function. Disable that check and you'll see all the developers log messages appearing. It can also be done by commenting out the log line in that place, which means that it looks like an empty function. You can add in the log line there.

Who Writes Code Like This?

The compiler. You'll see lots of cases where the code makes no sense. Where there are bizarre patterns of code being evaluated multiple times or where temporary variables are introduced and used even though there are more efficient ways of doing things. Just remember its not hand-written code and basically the compiling taking things like `if (s.length() != 0) return s.length();` and doing it step by step.