Android容易被逆向，对抗措施：加固保护（就是加壳吧）。不会被别人重打包盗版，不会被人注入。虽然已经自带了代码混淆器，但是从编程都是可以直接猜出来的。android还有一个签名文件。

使用Android SDK提供的apkbuilder.bat生成未签名的apk安装文件

JBE 反编译APK

apktool 基于AndroidManifest.xml，classes.dex，resources.arsc是三个文件反编译。我们需要知道apktool的原理，即知道对这三个文件的解析，这样可以用来加固使得apktool出错或者对抗加固。

aapt 编译或者反编译资源文件（aapt d resources test.apk）

编译过程：

xml中所有的资源描述经过编译，每个资源项给一个四字节无符号整数id，对资源的引用实际上是对ID的引用。通过aapt对资源文件进行编译生成的R.java

apk文件格式：

Android中一个apk可能包含多个资源包，默认情况下都只有一个就是应用的包名所在的资源包。

META\_INF目录下：CERT.SF,生成每个文件相对

的密钥(包括mainfest.mf文件的SHA1)；MANIFEST.MF,数字签名信息（所有其他文件的SHA-1并base64编码后的值）；xxx.SF,这是JAR文件的签名文件,占位符xxx标识了签名者；xxx.RSA,保存对CERT.S的私钥签名和公钥.对RSA文件的查看：openssl pkcs7 -inform DER -in xxx.RSA -noout -print\_certs –text.

assets和res/raw资源被原装不动地打包进APK

AndroidMainfest.xml 图片：[http://bbs.pediy.com/thread-194206.htm](http://bbs.pediy.com/thread-194206.htm" \t "_blank) 如不使用apktool进行“反编译”，就是二进制的不可读xml。

Resources.arsc   图片：<http://upload-images.jianshu.io/upload_images/2780242-1a1a7b34ded78941.png?imageMogr2/auto-orient/strip%7CimageView2/2/w/1240> 学习：<http://www.jianshu.com/p/3cc131db2002> 以及 <http://blog.csdn.net/jiangwei0910410003/article/details/50628894> ARSC文件作为Android应用资源的索引文件，其所包含的信息本质上是一张资源ID到资源的关系表（除了assets资源之外，其它的资源都会被赋予一个资源ID，屏幕设备尺寸不同时，app通过同样的ID但却能找到不同的资源进行显示） 。文件格式是由一系列的chunk构成,每一个chunk均包含如下结构的ResChunk\_header,用来描述这个chunk的基本信息：**资源索引表的头部信息**Resources.arsc文件的第一个结构是资源索引表头部，描述了Resources.arsc文件的大小和资源包数量；**资源项的值池** 定义了字符串的信息，使用个数、字符串类型、字符串相对于头部的位置 ; **Package数据块** 记录这个包的信息，一般来讲一个apk中只有一个资源包，没准还是会有多个的，其中有着类型字符串相对头部的偏移以及资源项名称字符串相对头部的偏移 ; **类型规范数据快** 每一种类型都对应有一个类型规范数据块，知道了一个资源项的配置状况之后，Android资源管理框架在检测到设备的配置信息发生变化之后，就可以知道是否需要重新加载该资源项 存有资源的类型，但是使用了Type ID来描述; **资源类型项数据块** 类型资源项数据块用来描述资源项的具体信息 一个具有n个配置的类型一共对应有n个类型资源项数据块　； **Type Spec与Config List**，通过一个资源ID找到不同资源文件的关键; **ResTable\_entry** 存储了资源项的值和资源项的资源ID，这里就是资源项的数据块。这种格式很明显，不能直接用文本编辑器打开。NDK中的aapt会生成一个resources.arsc文件和一个R.java，前者保存的是一个资源索引表，后者定义了各个资源ID常量，供在代码中索引资源。每一块都有相对于xx头部的偏移。

R.java Package ID相当于是一个命名空间，限定资源的来源。Android系统当前定义了两个资源命令空间，其中一个系统资源命令空间，它的Package ID等于0x01，另外一个是应用程序资源命令空间，它的Package ID等于0x7f

我们用工具反编译一些APP的时候，会看到一个smali文件夹，里面其实就是每个Java类所对应的smali文件

classes.dex 使用JDK提供的javac.exe编译.java类文件生成class文件-》使用Android SDK提供的dx.bat命令行脚本生成classes.dex文件

apk上传到应用宝以后会造成修改：

在lib目录中增加几个so文件

修改了AndroidManifest.xml、classes.dex、其他xml文件

tencent\_stub 这个文件中的值看起来是一种digest，曾经有人尝试过拖腾讯的壳：<https://www.52pojie.cn/thread-330022-1-1.html>

**加固程序**

Manifest.xml保留、增加入口点类（清单文件中增加入口点）、Native执行；

加壳（估计篡改一些指令）会导致ARM ELF头部被破坏，是的IDA Pro打开的时候崩溃，但是肯定还是有软件可以打开的，可以进行手工去壳。

花指令的push与pop增加了反汇编的难度？被认成函数

隐匿 APK 中的文件

加密APK中的文件（或者直接加壳）

**基于ID：** <http://www.cnblogs.com/wanyuanchun/p/4084292.html> android系统对于非法的res ID号是不会解析的。所以我们添加这个无用的属性后，并不影响该APK的正常工作(上图左下角所示)，但是对于apktool之类的逆向工具而言，他们却会对这个无用的属性进行解析(上图右下角所示)。所以，如果我们进行重打包的话，apktool就会将该属性变更为一个ID号0x01010003的可以被系统解析的属性。这样造成的后果就是：由于我们的APK中并没有实现trap.class类。或者，我们可以在其他tag中插入一些不会影响apk运行的属性(即新添加的属性不可被系统识别，重打包后该属性能被系统识别但又不会影响apk的运行)，然后在代码中检查AndroidManifest.xml是否含有该属性，如果有就说明软件被重打包了。

**重打包的过程：改变dex文件以后重新进行apk的签名**

apktool d xxx.apk out

重打包： 将dex解码以后变成smal（Icelandic中的组装器）的Dalvik VM比特代码格式。加入恶意代码并修改mainfest文件以后重新组装dex，编码mainfest文件，并进行签名。

apktool b    build命令

签名：使用openssl+SignApk.jar或者keytool+jarsign

**Android中是允许使用多个keystore对apk进行签名的**

**jarsign工具签名时使用的是keystore文件**

**signapk工具签名时使用的是pk8（私钥文件）,x509.pem（公钥文件）文件**

删除掉RSA文件以后重新使用自己的keystore进行签名，重新产生3个文件。重打包由于不知道私钥，所以至少要删除RSA文件，当然可以直接删除掉MAINFEST文件，都是需要重新生成的。