13 Android App 加固原理与技术浅析

一、前言

Java代码是非常容易反编译的，作为一种跨平台的、解释型语言，Java 源代码被编译成中间“字节码”存储于class文件中。由于跨平台的需要，

这些字节码带有许多的语义信息，很容易被反编译成Java源代码。为了很好地保护Java源代码，守护app的安全，解决移动App安全风险问题，

发展出了混淆代码、整体Dex加固、拆分 Dex 加固、虚拟机加固等方面保护手段，今天就抛砖引玉跟了解一下这些安全手段

二、混淆代码

混淆就是对发布出去的程序进行重新组织和处理，使得处理后的代码与处理前代码完成相同的功能，而混淆后的代码很难被反编译，即使反编译成功也很难得出程序的真正语义。

ProGuard就是一个混淆代码的开源项目，能够对字节码进行混淆、缩减体积、优化等处理。Proguard处理流程图如下所示，包含压缩、优化、混淆、预检四个主要环节：

1. 压缩（Shrink）：检测并移除代码中无用的类、字段、方法和特性（Attribute）；
2. 优化（Optimize）：对字节码进行优化，移除无用的指令。优化代码，非入口节点类会加上private/static/final，没有用到的参数会被删除，一些方法可能会变成内联代码；
3. 混淆（Obfuscate）：使用a、b、c、d这样简短而无意义的名称，对类、字段和方法进行重命名；
4. 预检（Preveirfy）：在Java平台上对处理后的代码进行预检，确保加载的class文件是可执行的。

混淆确实是有用处的，但混淆后的逻辑仍然可以看懂，但是如果你耐心一点任然能够追踪一些细节逻辑，而且国外也有些工具如DEGUADR，

它能够通过统计的方式来解混淆。虽然这个工具的正确率达不到100%，但是能在一定程度上帮助反编译代码。

三、Dex加固

3.1 加固的作用

加固的作用主要是提高软件的逆向成本,降低软件被破解的几率。APP加固可以
1、防止APP被逆向分析,反编译,二次打包,嵌入恶意代码
2、防止恶意攻击或竞争对手运用逆向工具，反编译为可读写的应用代码
3、防止攻击者使用静态分析工具与动态分析工具，分析并调试应用程序的运行逻辑，突破程序原来限制
4、防止攻击者恶意篡改代码及资源文件，进行盗版或植入广告等二次打包和重签名行为

3.2 加固原理

3.2.1 Dex文件基础知识

DEX是Dalvik EXecutable的简称，是打包.class文件为单一DEX文件并运行于Dalvik虚拟机，DEX文件打包进APK文件中（本质上是jar或zip文件）。

Dex文件整体结构说明：

数据名称	解释
dex_header	dex文件头部记录整个dex文件的相关属性
string_table	字符串数据索引，记录了每个字符串在数据区的偏移量
type_table	类似数据索引，记录了每个类型的字符串索引
proto_table	原型数据索引，记录了方法声明的字符串，返回类型字符串，参数列表
field_table	字段数据索引，记录了所属类，类型以及方法名
method_table	类方法索引，记录方法所属类名，方法声明以及方法名等信息
class_def	类定义数据索引，记录指定类各类信息，包括接口，超类，类数据偏移量
data_section	数据区，保存了各个类的真是数据

下面是DEX文件目录：

这里面，有3个成员我们需要特别关注，这在后面加固里会用到，它们分别是checksum、signature和fileSize。

checksum字段

checksum是校验码字段，占4bytes，主要用来检查从该字段（不包含checksum字段，也就是从12bytes开始算起）开始到文件末尾，这段数据是否完整，也就是完整性校验。它使用alder32算法校验。

signature字段

signature是SHA-1签名字段，占20bytes，作用跟checksum一样，也是做完整性校验。之所以有两个完整性校验字段，是由于先使用checksum字段校验可以先快速检查出错的dex文件，然后才使用第二个计算量更大的校验码进行计算检查。

fileSize字段

占4bytes，保存classes.dex文件总长度。

第一、二代加固就是将要保护的源程序apk 通过加密手段将dex程序保护起来，然后通过壳程序将dex文件解密，然后加载处理，不让攻击者获取真实的源dex，达到保护的目的

app加固技术随着技术的发展，经历了好几代技术变更，下面带大家一起了解一下。

四、加固技术发展历程

传统App加固技术，前后经历了四代技术变更，保护级别每一代都有所提升，但其固有的安全缺陷和兼容性问题始终未能得到解决。

而新五代加固技术—虚机源码保护，适用代码类型更广泛，App保护级别更高，兼容性更强，堪称未来级别的保护方,下面就分别对这些技术实现进行解析。

4.1 第一代加固技术-动态加载

第一代加固技术涉及三个对象，分别如下：

- 源程序
  源程序也就是我们的要加固的对象，这里面主要修改的是原apk文件中的classes.dex文件和AndroidManifest.xml文件。
- 壳程序
  壳程序主要用于解密经过加密了的dex文件，并加载解密后的原dex文件，并正常启动原程序。
- 加密程序
  加密程序主要是对原dex文件进行加密，加密算法可以是简单的异或操作、反转、rc4、des、rsa等加密算法。

加固过程可以分为如下4个阶段：

（1）加密阶段

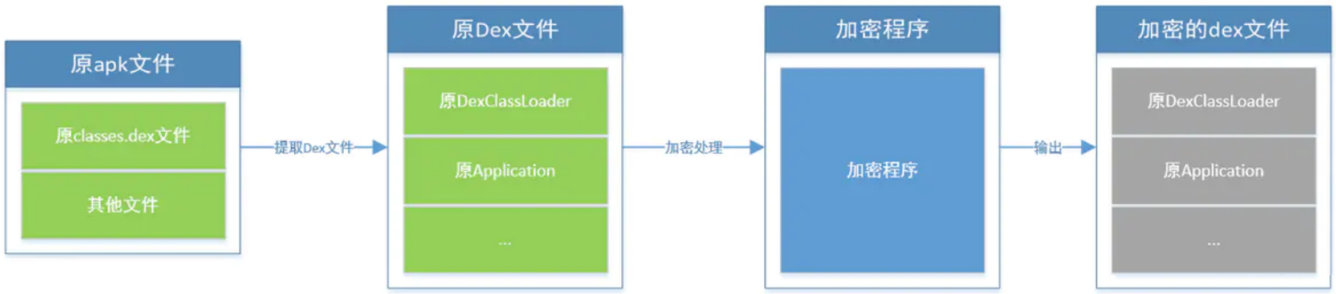
（2）合成新的dex文件

（3）修改原apk文件并重打包签名

（4）运行壳程序加载原dex文件

4.1.1、加密阶段

Dex的加密大致如下：



加密程序主要是对原dex文件进行加密，加密算法可以是简单的异或操作、反转、rc4、des、rsa等加密算法。

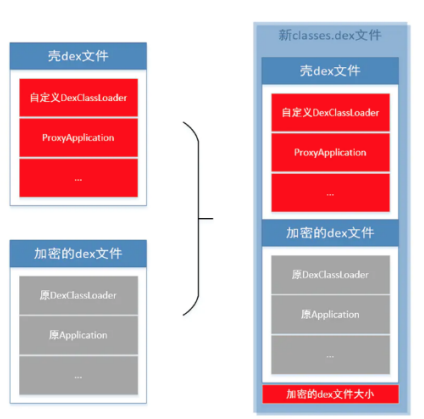
4.1.2、合成新的dex文件

将一个文件(加密之后的源Apk)写入到壳Dex中，那么需要修改壳Dex的文件校验码(checksum).因为它是会检查文件是否有错误。那么signature也是一样，也是唯一识别文件的算法。

还有就是需要修改脱壳dex文件的大小。此外，还需要一个操作，就是标注一下加密的Apk的大小，因为在脱壳的时候，需要知道加密后的源Apk的大小，才能正确的得到加密后的源Apk。

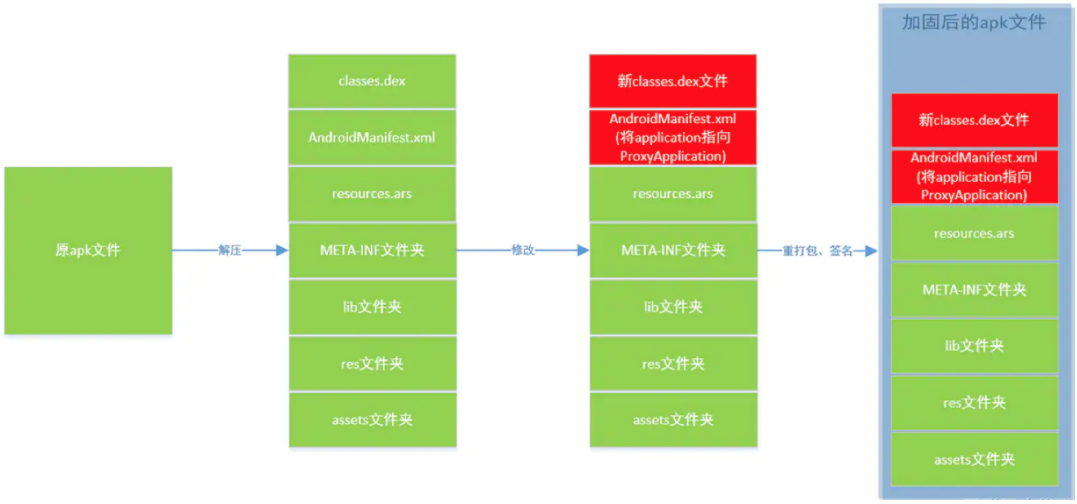
这个值直接放到文件的末尾就可以了。这样，就生成了一个壳dex文件。（脱壳dex追加源apk、源apk大小，并修改脱壳dex头部，从而生成了壳dex文件。）

即：修改Dex的三个文件头，将源Apk的大小追加到壳dex的末尾就可以了。修改之后得到新的Dex文件样式如下：核心代码课参见Mymain.java




```
File payloadSrcFile = new File("F:\\AndroidStudioProjects\\sourceapk\\app\\release\\app-release.apk");    //需要加壳的程序
System.out.println("apk size:"+payloadSrcFile.length());
File unShellDexFile = new File("F:\\AndroidStudioProjects\\unshell\\app\\release\\classes.dex");    //解壳dex
byte[] payloadArray = encrpt(readFileBytes(payloadSrcFile));//以二进制形式读出apk，并进行加密处理
byte[] unShellDexArray = readFileBytes(unShellDexFile);//以二进制形式读出dex
int payloadLen = payloadArray.length;
int unShellDexLen = unShellDexArray.length;
int totalLen = payloadLen + unShellDexLen +4;//多出4字节是存放长度的
byte[] newdex = new byte[totalLen]; // 申请了新的长度
//添加解壳DEX
System.arraycopy(unShellDexArray, 0, newdex, 0, unShellDexLen);//先拷贝dex内容
//添加加密后的APK
System.arraycopy(payloadArray, 0, newdex, unShellDexLen, payloadLen);//再在dex内容后面拷贝apk的内容
//添加APK长度
System.arraycopy(intToByte(payloadLen), 0, newdex, totalLen-4, 4);//最后4字节为长度
//修改DEX file size文件头
fixFileSizeHeader(newdex);
//修改DEX SHA1 文件头
fixSHA1Header(newdex);
//修改DEX CheckSum文件头
fixCheckSumHeader(newdex);
```

4.1.3 修改原apk文件并重打包签名

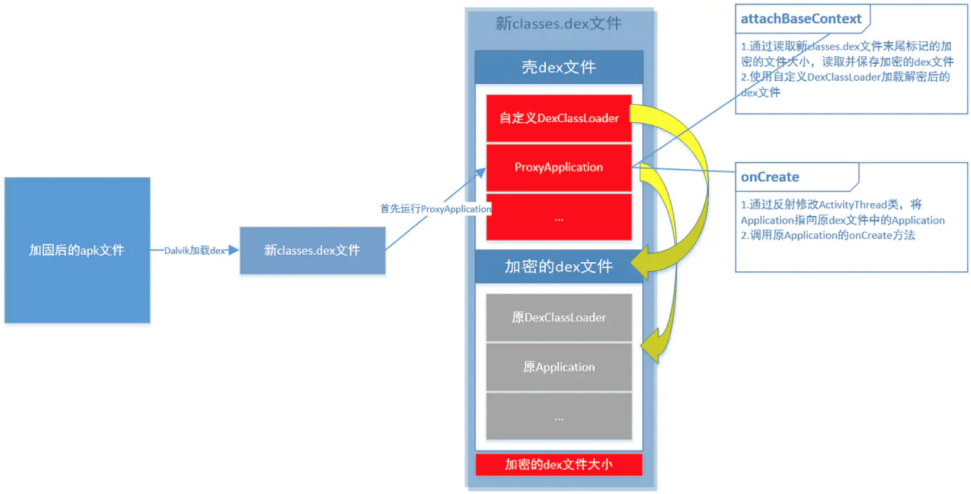


把解压后apk目录下原来的classes.dex文件替换成我们在上一步合成的新的classes.dex文件。然后，由于我们程序运行的时候，

首先加载的其实是壳程序里的ProxyApplication类。所以，我们需要修改AndroidManifest.xml文件，指定application为ProxyApplication，

这样才能正常找到识别ProxyApplication类并运行壳程序。

4.1.4 运行壳程序加载原dex文件




```
Log.i(TAG, msg: "进入attachBaseContext方法");
try {
    // 创建两个私有的、可写的文件目录payload_dex payload_lib
    File dex=getDir( name: "payload_dex",MODE_PRIVATE);
    File lib=getDir( name: "payload_lib",MODE_PRIVATE);
    dexPath=dex.getAbsolutePath();
    libPath=lib.getAbsolutePath();
    apkFileName=dex.getAbsolutePath()+File.separator+"shell.apk";
    File dexFile=new File(apkFileName);
    if (!dexFile.exists()){
        // 在payload_dex目录下创建shell.apk
        dexFile.createNewFile();
        // 读取classes.dex文件
        byte[] dexdata=readDexFileFromApk();
        // 分离出源APK文件
        splitPayloadFromDex(dexdata);
    }
    // 配置动态加载环境
    Object activityThreadObject=RefinvokeMethod.invokeStaticMethod( class_name: "android.app.ActivityThread",
d_name: "currentActivityThread",new Class[]{},new Object[]{});
    // 获取APK包名
    String packageName = getPackageName();
    ArrayMap packages= (ArrayMap) RefinvokeMethod.getField( class_name: "android.app.ActivityThread",activityThreadObject,
ame: "mPackages");
    WeakReference weakReference= (WeakReference) packages.get(packageName);
    // 创建源APK的DexClassLoader对象，加载APK的类和C/C++代码
    ClassLoader classLoader= (ClassLoader) RefinvokeMethod.getField( class_name: "android.app.LoadedApk",weakReference.get(),
ame: "mClassLoader");
    DexClassLoader dexClassLoader=new DexClassLoader(apkFileName,dexPath,libPath,classLoader);
    // 把当前进程的DexClassLoader设置为源APK的DexClassLoader
    RefinvokeMethod.setField( class_name: "android.app.LoadedApk", field_name: "mClassLoader",weakReference.get(),dexClassLoader);
    try {
        Object objectMain = dexClassLoader.loadClass( name: "com.example.sourceapk.MainActivity");
        Log.i(TAG, msg: "MainActivity类加载完毕");
    } catch (ClassNotFoundException e) {
```

```
Log.i(TAG, msg: "进入onCreate方法");                                                    25  6
String applicationName="";
ApplicationInfo ai=null;
try {
    ai=getPackageManager().getApplicationInfo(getPackageName(), PackageManager.GET_META_DATA);
    applicationName=ai.metaData.getString( key: "ApplicationName");
} catch (PackageManager.NameNotFoundException e) {...}
Object activityThreadObj=RefinvokeMethod.invokeStaticMethod( class_name: "android.app.ActivityThread",
d_name: "currentActivityThread",new Class[]{},new Object[]{});
Object mBoundApplication=RefinvokeMethod.getField( class_name: "android.app.ActivityThread",activityThreadObj,
ame: "mBoundApplication");
Object info=RefinvokeMethod.getField( class_name: "android.app.ActivityThread$AppBindData",mBoundApplication, field_name: "info");
// 将当前进程的mApplication设置为null
RefinvokeMethod.setField( class_name: "android.app.LoadedApk", field_name: "mApplication",info, value: null);
Object mInitApplication=RefinvokeMethod.getField( class_name: "android.app.ActivityThread",activityThreadObj,
ame: "mInitialApplication");
ArrayList<Application> mAllApplications= (ArrayList<Application>) RefinvokeMethod.getField( class_name: "android.app.ActivityThread",
ityThreadObj, field_name: "mAllApplications");
mAllApplications.remove(mInitApplication);
ApplicationInfo mApplicationInfo= (ApplicationInfo) RefinvokeMethod.getField( class_name: "android.app.LoadedApk",info,
ame: "mApplicationInfo");
ApplicationInfo appinfo= (ApplicationInfo) RefinvokeMethod.getField( class_name: "android.app.ActivityThread$AppBindData",
dApplication, field_name: "appInfo");
mApplicationInfo.className=applicationName;
appinfo.className=applicationName;
// 执行makeApplication(false,null)
Application app= (Application) RefinvokeMethod.invokeMethod( class_name: "android.app.LoadedApk", method_name: "makeApplication",info,new
[]{boolean.class, Instrumentation.class},new Object[]{false,null});
RefinvokeMethod.setField( class_name: "android.app.ActivityThread", field_name: "mInitialApplication",activityThreadObj,app);
ArrayMap mProviderMap= (ArrayMap) RefinvokeMethod.getField( class_name: "android.app.ActivityThread",activityThreadObj,
ame: "mProviderMap");
Iterator iterator=mProviderMap.values().iterator();
while (iterator.hasNext()){
    Object mProviderClientRecord=iterator.next();
    Object mLocalProvider=RefinvokeMethod.getField( class_name: "android.app.ActivityThread$ProviderClientRecord",mProviderClientRecord,
ame: "mLocalProvider");
    RefinvokeMethod.setField( class_name: "android.content.ContentProvider", field_name: "mContext",mLocalProvider,app);
```

在attachBaseContext方法里，主要做两个工作：

- 读取classes.dex文件末尾记录加密dex文件大小的数值，则加密dex文件在新classes.dex文件中的位置为：len(新classes.dex文件) - len(加密dex文件大小)。
- 将加密的dex文件读取出来，加密并保存到资源目录下然后使用自定义的DexClassLoader加载解密后的原dex文件

在onCreate方法中，主要做两个工作：

- 通过反射修改ActivityThread类，并将Application指向原dex文件中的Application
- 创建原Application对象，并调用原Application的onCreate方法启动原程序

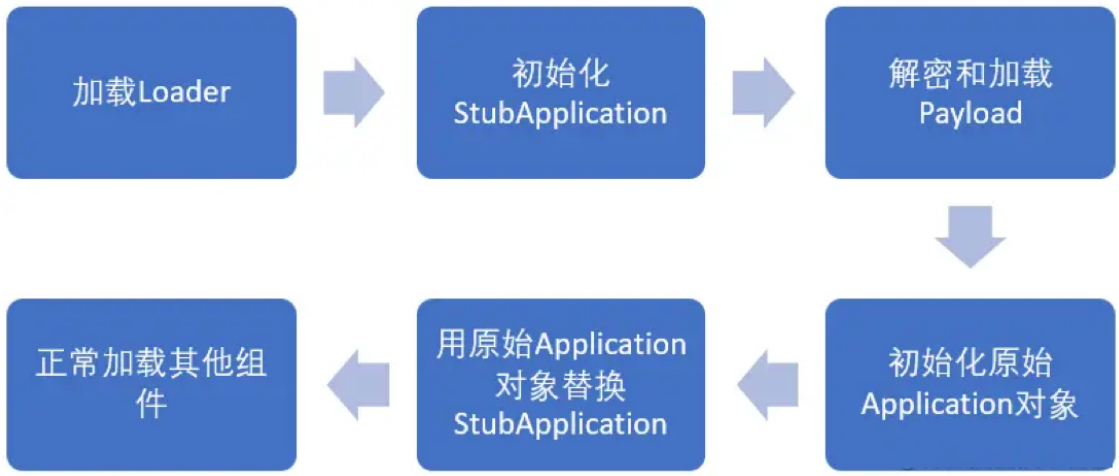
更多源可以参考文件ProxyApplication.java。

优势：
　dex加密保存
不足：
　dex关键部分逻辑必须解压在文件系统中，通过自定义虚拟机、截获关键函数，在加载dex文件时比解密后的内容复制或者使用root机器可以获取到解密后的dex文件
破解状况：
　目前基本上已被破解，部分反编译工具已集成自动脱壳功能。
实例：
　早期的爱加密版本

4.2 第二代加固技术-不落地加载

相对第一代加固技术，第二代加固技术在APK修改方面已经完善，能做到对开发的零干扰。开发过程中不需要对应用做特殊处理，

只需要在最终发布前进行保护即可。而为了实现这个零干扰的流程，Loader需要处理好Android的组件的生命周期。
主要流程：
1）Loader被系统加载。
2）系统初始化Loader内的StubApplication。
3）StubApplication解密并且加载原始的DEX文件（Payload）。
4）StubApplication从原始的DEX文件（Payload）中找到原始的Application对象，创建并初始化。
5）将系统内所有对StubApplication对象的引用使用替换成原始Application，此步骤使用JAVA的反射机制实现。
6）由Android系统进行其他组件的正常生命周期管理。



不落地加载技术是在第一代加固技术的基础上改进，主要解决第一代技术中Payload必须释放到文件系统（俗称落地）的缺陷，其主要的技术方案有两种：

4.2.1 透明加解密方案

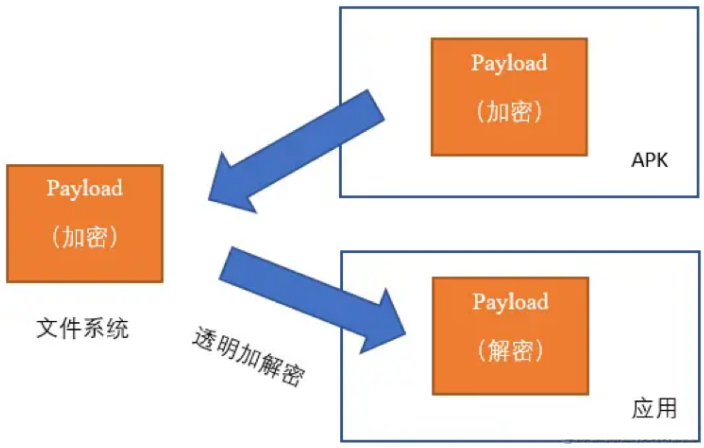
拦截系统IO相关的函数（如read、write），在这些函数中提供透明加解密。具体的流程是：
1）关键逻辑（Payload）以加密的方式存储在APK中。
2）运行时加载部分（Loader）将关键逻辑释（Payload）放到文件系统，此时关键逻辑（Payload）还处于加密状态。
3）加载部分拦截对应的系统IO函数（read，write等）。
4）加载部分（Loader）正常调用Java动态加载机制。由于虚拟机的IO部分被拦截，所以虚拟机读取到已经解密的关键逻辑（Payload）

加密具体流程是：

1）关键逻辑（Payload）以加密的方式存储在APK中。

2）运行时加载部分（Loader）将关键逻辑释（Payload）放到内存。

3）加载部分调用虚拟机内部接口进行加载。



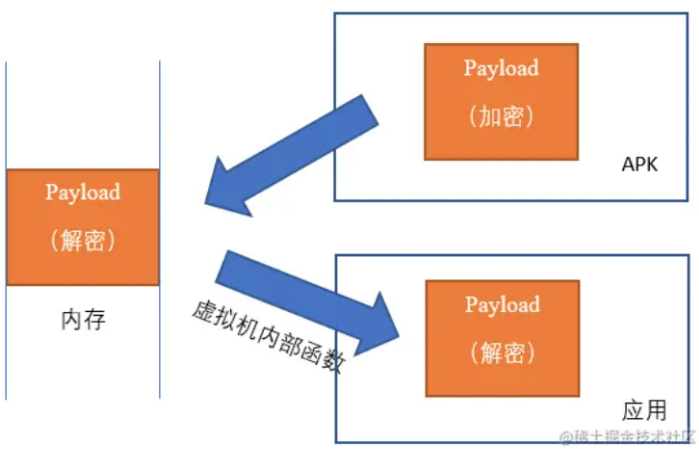
4.2.1 调用虚拟机提供的函数进行不落地的加载方案

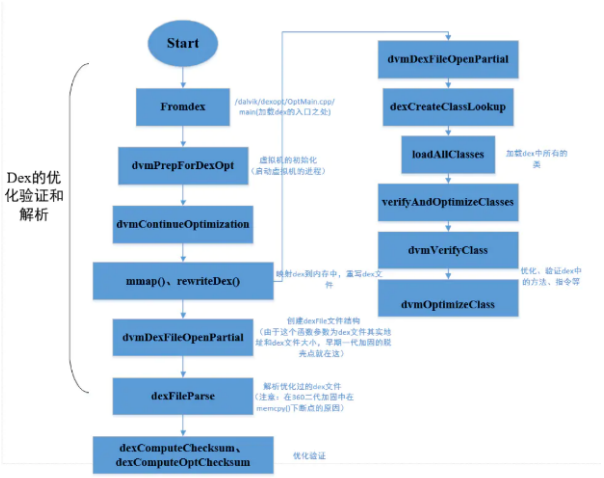
直接调用虚拟机提供的函数进行不落地的加载，具体流程是：

1）关键逻辑（Payload）以加密的方式存储在APK中。

2）运行时加载部分（Loader）将关键逻辑释（Payload）放到内存。

3）加载部分调用虚拟机内部接口进行加载





关键的系统函数如下：

dvmDexFileOpenFromFd 从文件描述符获取DexFile结构体

dvmDexFileOpenPartial 从内存获取DexFile结构体

兼容性

方案1透明加密方案由于其需要拦截系统的IO函数，这部分会使用inline hook或者got hook等技术，其会带来一定的兼容性问题

方案2的不落地加载方案由于其调需要调用系统内部的接口，而这个接口并不导出，各个厂商在实现时又有各自的自定义修改，导致该方案存在兼容性问题。

缺陷与对抗

不落地加载方案由于其调需要调用系统内部的接口，而这个接口并不导出，各个厂商在实现时又有各自的自定义修改，导致该方案存在兼容性问题。
第二代加固技术在应用启动时要处理大量的加解密加载操作，会造成应用长时间假死（黑屏），用户体验差。
在加固技术实现上没有本质区别，虽然能防止第一代加固技术文件必须落地被复制的缺陷，但是也可以从以下方面进行对抗：

例如内存中的DEX文件头会被清除，用于防止在dump文件中被找到；DEX文件结构被破坏，例如增加了一些错误的数据，提高恢复的成本。
但是Payload被加载之后，在内存中是连续的，利用gdb等调试工具dump内存后可以直接找到Payload，进行简单的处理之后可以恢复出100%的Payload文件。
和第一代加固技术的对抗方法一样，不落地加载也无法对抗自定义虚拟机。只需对上述的关键函数进行拦截然后将对应的内存段写出去，即可恢复Payload。注意，由于IO相关的函数被拦截，所以无法直接调用read/write等函数进行直接的读写，需要使用syscall函数进行绕过。
虽然厂商会自己实现可能上述函数，从而绕过上述函数的拦截。但是Android的类加载器必须能找到对于的结构体才能正常执行，攻击者可以以类加载器做为起点，找到对应的Payload在内存中的位置。

优势：
当前市面最常见、通常作为一项基础性的免费服务向用户提供

破解状况：
已经出现专业人士自行研究的手工脱壳方法（Dexhunter），但尚未出现自动脱壳工具，破解难度仍然很大。
实例：
市面上流行的大多数在线加固服务，如腾讯乐固，360加固，百度加固等

4.3 第三代加固技术-指令抽离

由于第二代加固技术仅仅对文件级别进行加密，其带来的问题是内存中的Payload是连续的，可以被攻击者轻易获取。第三代加固技术对这部分进行了改进，将保护级别降到了函数级别。

主要的流程是：发布阶段将原始DEX内的函数内容（Code Item）清除，单独移除到一个文件中。

运行阶段将函数内容重新恢复到对应的函数体。恢复的时间点有几个方式：

A、加载之后恢复函数内容到DEX壳所在的内存区域。

B、加载之后将函数内容恢复到虚拟机内部的结构体上：

虚拟机读取DEX文件后内部对每一个函数有一个结构体，这个结构体上有一个指针指向函数内容（CodeItem），可以通过修改这个指针修改对应的函数内容。

C、拦截虚拟机内与查找执行代码相关的函数，返回函数内容。

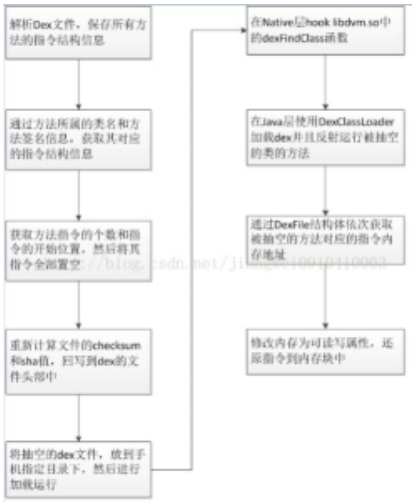
总结一下流程：

第一、抽取指令流程

- 1、解析原始dex文件格式，保存所有方法的代码结构体信息。
- 2、通过传入需要置空指令的方法和类名，检索到其代码结构体信息。
- 3、通过方法的代码结构体信息获取指令个数和偏移地址，构造空指令集，然后覆盖原始指令。
- 4、重新计算dex文件的checksum和signature信息，回写到头部信息中。

第二、指令还原流程

- 1、native层hook系统函数dexFindClass，获取类结构体信息。
- 2、获取类中所有的方法信息，通过指定方法名进行过滤，获取该方法的代码结构体信息。
- 3、获取该方法被抽取的指令集，修改方法对应的内存地址为可读属性，直接进行指令还原。



兼容性

指令抽离技术使用了大量的虚拟内部结构与未被文档的特性，再加上Android复杂的厂商定制，带来大量的兼容性问题。

缺陷与对抗

指令抽离技术的某些方案与虚拟机的JIT性能优化冲突，无法达到最佳的运行性能。依旧使用了java虚拟机进行函数内容的执行。攻击者可以通过自定义Android虚拟机，在解释器的代码上做记录一个函数的内容（CodeItem）。接下来遍历触发所有函数，从而获取到全部的函数内容。最终重新组装成一个完整的DEX文件。目前已经有自动化工具可以指令抽离技术中脱壳。

第三代加固DEX文件脱壳流程

优势：
当前市面最常见、通常作为一项基础性的免费服务向用户提供

破解状况：

部分被破解，已经出现专业人士自行研究的手工脱壳方法，但尚未出现自动脱壳工具

实例：
现在免费版“爱加密”，梆梆安全免费版

4.4.第四代加固技术-指令转换/VMP（虚拟机保护）

第三代加固技术在函数级别的保护，使用Android虚拟机内的解释器执行代码，带来可能被记录的缺陷，第四代加固技术，在抽取代码后，并没有对代码进行还原，而是使用自己的解释器来避免第三代的缺陷。

而自定义的解释器无法对Android系统内的其他函数进行直接调用，必须使用JAVA的JNI接口进行调用。其主要实现由两种：

A、DEX文件内的函数被标记为native，内容被抽离并转换成一个符合JNI要求的动态库。动态库内通过JNI和Android系统进行交互。

B、DEX文件内的函数被标记为native，内容被抽离并转换成自定义的指令格式，该格式使用自定义接收器执行，和A一样需要使用JNI和Android系统进行调用。

兼容性

第四代VMP加固技术一般配合第三代加固技术使用，所以第三代的所有兼容性问题，指令转换/VMP加固也存在。

缺陷与对抗

不论使用指令转换/VMP加固的A方案或者B方案，其必须通过虚拟机提供的JNI接口与虚拟机进行交互，攻击者可以直接将指令转换/VMP加固方案当作黑盒，通过自定义的JNI接口对象，

对黑盒内部进行探测、记录和分析，获得加固的操作码，进而得到完整DEX程序。

另外，第四代VMP加固技术只实现Java代码保护，没有做到使用VMP技术来保护C/C++等代码，安全保护能力有所欠缺。

破解状况：

部分被破解，已经出现专业人士自行研究的手工脱壳方法，但尚未出现自动脱壳工具

实例：
大部分需要定制收费的加密服务（如爱加密，梆梆安全，中国移动加固，以及部分手机银行自行研究的加固等）

4. 5 第五代加固技术-虚拟机源码保护

虚机源码保护加固是用虚机技术保护所有的代码，包括Java，Kotlin，C/C++，Objective-C，Swift等多种代码，具备极高的兼容性；使App得到更高安全级别的保护，运行更加稳定。

虚机源码保护为用户提供一套完整的工具链，首先把用户待保护的核心代码编译成中间的二进制文件，随后生成独特的虚机源码保护执行环境和只能在该环境下执行的运行程序。

虚机源码保护会在App内部隔离出独立的执行环境，该核心代码的运行程序在此独立的执行环境里运行。即便App本身被破解，这部分核心代码仍然不可见。

破解现状：

大多数未被破解

实例：
极为少数，需要特殊定制的加固服务，通常用于银行金融机构等关乎国家安全的重点领域

五、参考学习资料

Android加固和脱壳原理浅论 https://www.52pojie.cn/thread-977325-1-1.html

Android DEX安全攻防战 https://www.pianshen.com/article/9877168280/

安卓App加固技术发展历程 https://mp.weixin.qq.com/s/RONlZf0B60zMw5_-B-tpmA

加固技术发展与对抗 https://juejin.cn/post/6844904176070164488

一文了解安卓APP逆向分析与保护机制 https://mp.weixin.qq.com/s/fG - 0syEtdrXgSVtHzCzfA

百度加固逆向分析—dex还原--二代抽取壳 https://blog.csdn.net/zhangmiaoping23/article/details/102678531

入门级加固--3种加固方式学习记录 https://www.52pojie.cn/thread-1043762-1-1.html

DexClassLoader和PathClassLoader载入Dex流程 https://www.cnblogs.com/yfceshi/p/7230522.htmlhttps://www.jianshu.com/p/e4d2091bbf9f

Android应用实现「类方法指令抽取方式」加固方案原理解析 https://mp.weixin.qq.com/s/Qs7AZswzpHZlv5cNWxztkg

App加固的种类甄别与侦查 https://mp.weixin.qq.com/s/esDosyMxOotYw5ps2xGiRA

Android逆向-从入门到放弃到受益 https://mp.weixin.qq.com/s/T80XG8ZnyN8LrGP0aexxgA