VasDolly实现原理

Jump to bottom

yanyongshan edited this page on Sep 7, 2020 · 5 revisions

概述

众所周知，因为国内Android应用分发市场的现状，我们在发布APP时，一般需要生成多个渠道包，上传到不同的应用市场。这些渠道包需要包含不同的渠道信息，在APP和后台交互或者数据上报时，会带上各自的渠道信息。这样，我们就能统计到每个分发市场的下载数、用户数等关键数据。

普通的多渠道打包方案

既然我们需要进行多渠道打包，那我们就看下最常见的多渠道打包方案。

Android Gradle Plugin

Gradle Plugin本身提供了多渠道的打包策略： 首先，在AndroidManifest.xml中添加渠道信息占位符：

```
<meta-data
android:name="InstallChannel" android:value="${InstallChannel}" />
```

然后，通过Gradle Plugin提供的`productFlavors`标签，添加渠道信息：

```
productFlavors{
    "YingYongBao"{
        manifestPlaceholders = [InstallChannel : "YingYongBao"]
    }
    "360"{
        manifestPlaceholders = [InstallChannel : "360"]
    }
}
```

这样，Gradle编译生成多渠道包时，会用不同的渠道信息替换AndroidManifest.xml中的占位符。我们在代码中，也就可以直接读取AndroidManifest.xml中的渠道信息了。

但是，这种方式存在一些缺点：

1. 每生成一个渠道包，都要重新执行一遍构建流程，效率太低，只适用于渠道较少的场景。
2. Gradle会为每个渠道包生成一个不同的BuildConfig.java类，记录渠道信息，导致每个渠道包的DEX的CRC值都不同。一般情况下，这是没有影响的。但是如果你使用了微信的Tinker热补丁方案，那么就需要为不同的渠道包打不同的补丁，这完全是不可以接受的。（因为Tinker是通过对比基础包APK和新包APK生成差分补丁，然后再把补丁和基础包APK一起合成新包APK。这就要求用于生成差分补丁的基础包DEX和用于合成新包的基础包DEX是完全一致的，即：每一个基础渠道包的DEX文件是完全一致的，不然就会合成失败）

ApkTool

ApkTool是一个逆向分析工具，可以把APK解开，添加代码后，重新打包成APK。因此，基于ApkTool的多渠道打包方案分为以下几步：

1. 复制一份新的APK
2. 通过ApkTool工具，解压APK（apktool d origin.apk）
3. 删除已有签名信息
4. 添加渠道信息（可以在APK的任何文件添加渠道信息）
5. 通过ApkTool工具，重新打包生成新APK（apktool b newApkDir）
6. 重新签名

经过测试，这种方案完全是可行的。

优点： 不需要重新构建新渠道包，仅需要复制修改就可以了。并且因为是重新签名，所以同时支持V1和V2签名。

缺点：

1. ApkTool工具不稳定，曾经遇到过升级Gradle Plugin版本后，低版本ApkTool解压APK失败的情况。
2. 生成新渠道包时，需要重新解包、打包和签名，而这几步操作又是相对比较耗时的。经过测试：生成企鹅电竞10个渠道包需要16分钟左右，虽然比Gradle Plugin方案减少很多耗时。但是若需要同时生成上百个渠道包，则需要几个小时，显然不适合渠道非常多的业务场景。

那有没有一种方案：可以在添加渠道信息后，不需要重新签名那？首先我们要了解一下APK的签名和校验机制。

数据摘要、数字签名和数字证书

在进一步学习V1和V2签名之前，我们有必要学习一下签名相关的基础知识。

数据摘要

数据摘要算法是一种能产生特定输出格式的算法，其原理是根据一定的运算规则对原始数据进行某种形式的信息提取，被提取出的信息就是原始数据的消息摘要，也称为数据指纹。 一般情况下，数据摘要算法具有以下特点：

1. 无论输入数据有多大（长），计算出来的数据摘要的长度总是固定的。例如：MD5算法计算出的数据摘要有128Bit。
2. 一般情况下（不考虑碰撞的情况下），只要原始数据不同，那么其对应的数据摘要就不会相同。同时，只要原始数据有任何改动，那么其数据摘要也会完全不同。即：相同的原始数据必有相同的数据摘要，不同的原始数据，其数据摘要也必然不同。
3. 不可逆性，即只能正向提取原始数据的数据摘要，而无法从数据摘要中恢复出原始数据。

著名的摘要算法有RSA公司的MD5算法和SHA系列算法。

数字签名和数字证书

数字签名和数字证书是成对出现的，两者不可分离（***数字签名主要用来校验数据的完整性，数字证书主要用来确保公钥的安全发放***）。

要明白数字签名的概念，必须要了解数据的加密、传输和校验流程。一般情况下，要实现数据的可靠通信，需要解决以下两个问题：

1. 确定数据的来源是其真正的发送者。
2. 确保数据在传输过程中，没有被篡改，或者若被篡改了，可以及时发现。

而数字签名，就是为了解决这两个问题而诞生的。 首先，数据的发送者需要先申请一对公私钥对，并将公钥交给数据接收者。 然后，若数据发送者需要发送数据给接收者，则首先要根据原始数据，生成一份数字签名，然后把原始数据和数字签名一起发送给接收者。 数字签名由以下两步计算得来：

1. 计算发送数据的数据摘要
2. 用私钥对提取的数据摘要进行加密

这样，数据接收者拿到的消息就包含了两块内容：

1. 原始数据内容
2. 附加的数字签名

接下来，接收者就会通过以下几步，校验数据的真实性：

1. 用相同的摘要算法计算出原始数据的数据摘要。
2. 用预先得到的公钥解密数字签名。
3. 对比签名得到的数据是否一致，如果一致，则说明数据没有被篡改，否则数据就是脏数据了。

因为私钥只有发送者才有，所以其他人无法伪造数字签名。这样通过数字签名就确保了数据的可靠传输。 综上所述，数字签名就是***只有发送者才能产生的别人无法伪造的一段数字串，这段数字串同时也是对发送者发送数据真实性的一个有效证明。***

想法虽好，但是上面的整个流程，有一个前提，就是数据接收者能够正确拿到发送者的公钥。如果接收者拿到的公钥被篡改了，那么坏人就会被当成好人，而真正的数据发送者发送的数据则会被视作脏数据。那怎么才能保证公钥的安全性那？这就要靠数字证书来解决了。

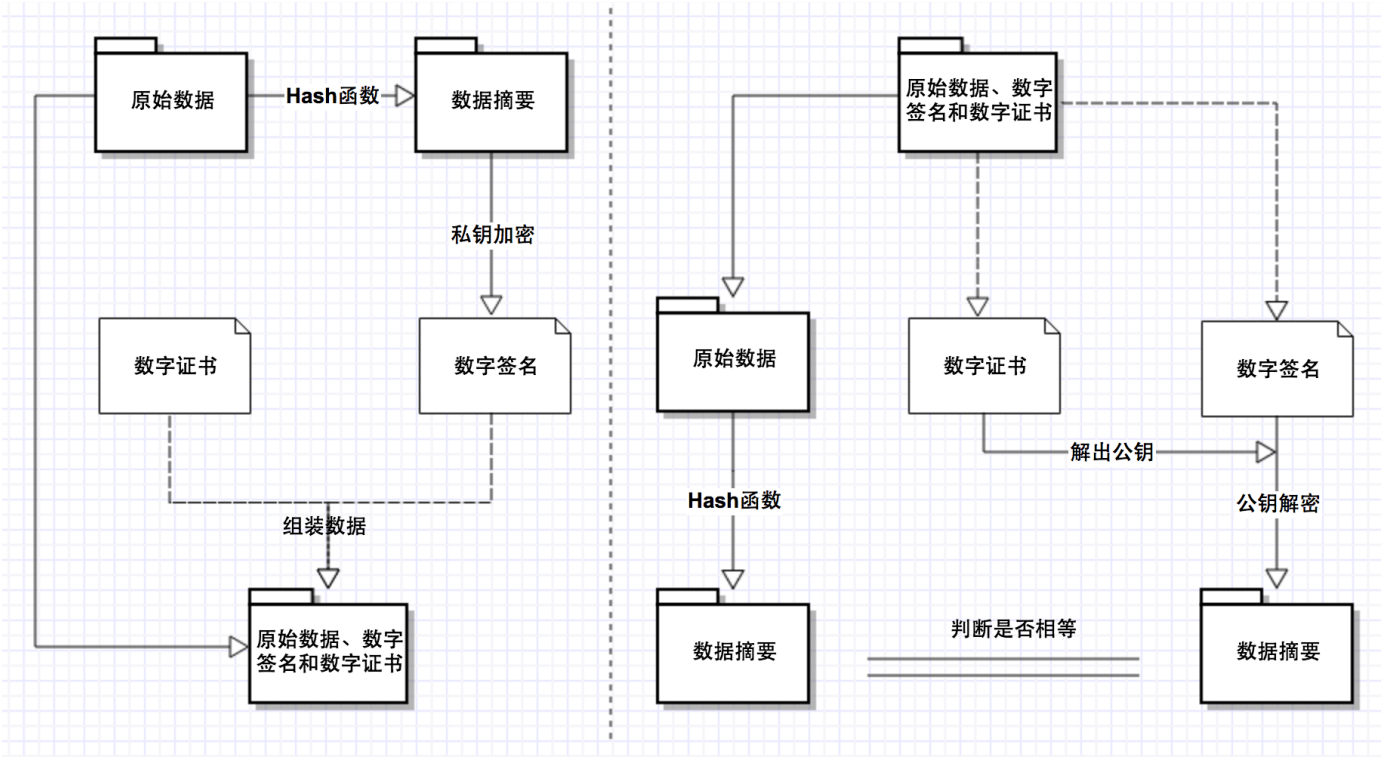
数字证书是由有公信力的证书中心（CA）颁发给申请者的证书，主要包含了：证书的发布机构、证书的有效期、申请者的公钥、申请者信息、数字签名使用的算法，以及证书内容的数字签名。

可见，数字证书也用到了数字签名技术。只不过签名的内容是数据发送方的公钥，以及一些其它证书信息。 这样数据发送者发送的消息就包含了三部分内容：

1. 原始数据内容
2. 附加的数字签名
3. 申请的数字证书。

接收者拿到数据后，首先会根据CA的公钥，解码出发送者的公钥。然后就与上面的校验流程完全相同了。

所以，**数字证书主要解决了公钥的安全发放问题。** 因此，包含数字证书的整个签名和校验流程如下图所示：



V1签名和多渠道打包方案

V1签名机制

默认情况下，APK使用的就是V1签名。解压APK后，在`META-INF`目录下，可以看到三个文件：MANIFEST.MF、CERT.SF、CERT.RSA。它们都是V1签名的产物。

其中，`MANIFEST.MF`文件内容如下所示：

```
Manifest-Version: 1.0
Created-By: 1.7.0_60-ea (Oracle Corporation)

Name: res/drawable/skin_drawable_btm_line.xml
SHA1-Digest: 5nATDWzhCP2Mz4XrmtWqxzZriLo=

Name: res/drawable/common_dialog_brand.xml
SHA1-Digest: ew1uX1u8XCg4L4CzmMVTHTRQG6g=
```

它记录了APK中所有原始文件的数据摘要的Base64编码,而数据摘要算法就是`SHA1`。

`CERT.SF`文件内容如下所示：

```
Signature-Version: 1.0
SHA1-Digest-Manifest-Main-Attributes: GQIRuJcAmolTl8SuRVbT9a+jZmw=
Created-By: 1.7.0_60-ea (Oracle Corporation)
SHA1-Digest-Manifest: v13tM8BG6wHiNs2r12Zr5SocH9k=

Name: res/drawable/skin_drawable_btm_line.xml
SHA1-Digest: o6FRcC/m/vZLlLBJro+YHMKHRsk=

Name: res/drawable/common_dialog_brand.xml
SHA1-Digest: B/WLRMSUVX+Gllj91AZRjaMgbow=
```

`SHA1-Digest-Manifest-Main-Attributes`主属性记录了`MANIFEST.MF`文件所有主属性的数据摘要的Base64编码。`SHA1-Digest-Manifest`则记录了整个`MANIFEST.MF`文件的数据摘要的Base64编码。其余的普通属性则和`MANIFEST.MF`中的属性一一对应，分别记录了对应数据块的数据摘要的Base64编码。例如：`CERT.SF`文件中skin_drawable_btm_line.xml对应的SHA1-Digest，就是下面内容的数据摘要的Base64编码。

```
Name: res/drawable/skin_drawable_btm_line.xml
SHA1-Digest: JqJbk6/AsWZMcGVehCXb33Cdtrk=
\r\n
```

这里要注意的是：**最后一行的换行符是必不可少，需要参与计算的。**

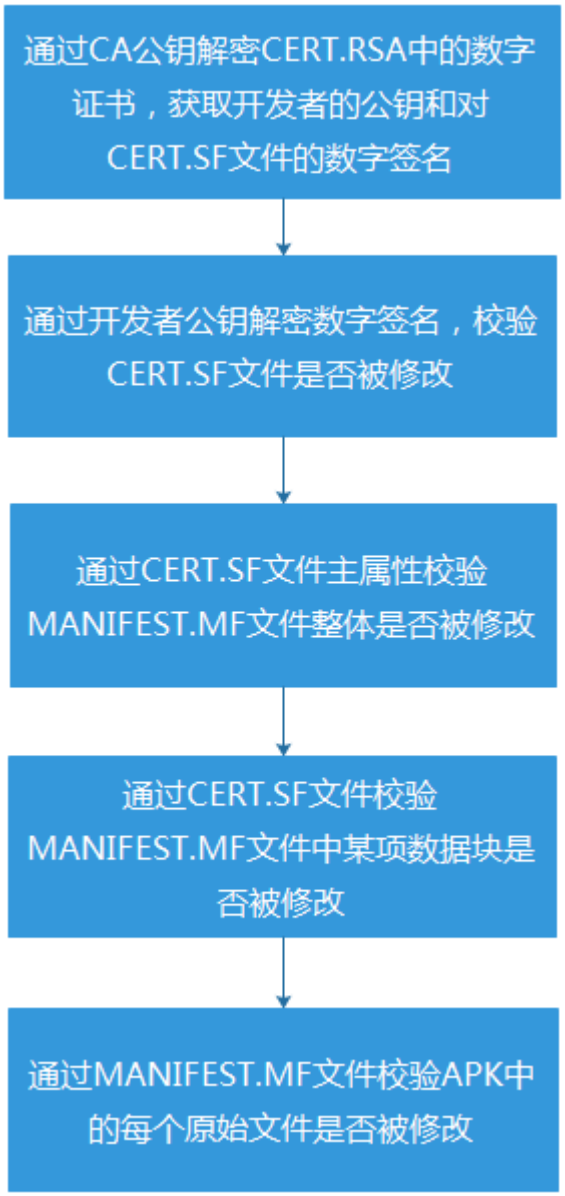
`CERT.RSA`文件包含了对`CERT.SF`文件的数字签名和开发者的数字证书。`RSA`就是计算数字签名使用的非对称加密算法。

V1签名的详细流程可参考SignApk.java，整个签名流程如下图所示：

整个签名机制的最终产物就是MANIFEST.MF、CERT.SF、CERT.RSA三个文件。

V1校验流程

在安装APK时，Android系统会校验签名，检查APK是否被篡改。代码流程是：`PackageManagerService.java` -> `PackageParser.java`，`PackageParser`类负责V1签名的具体校验。整个校验流程如下图所示：



若中间任何一步校验失败，APK就不能安装。

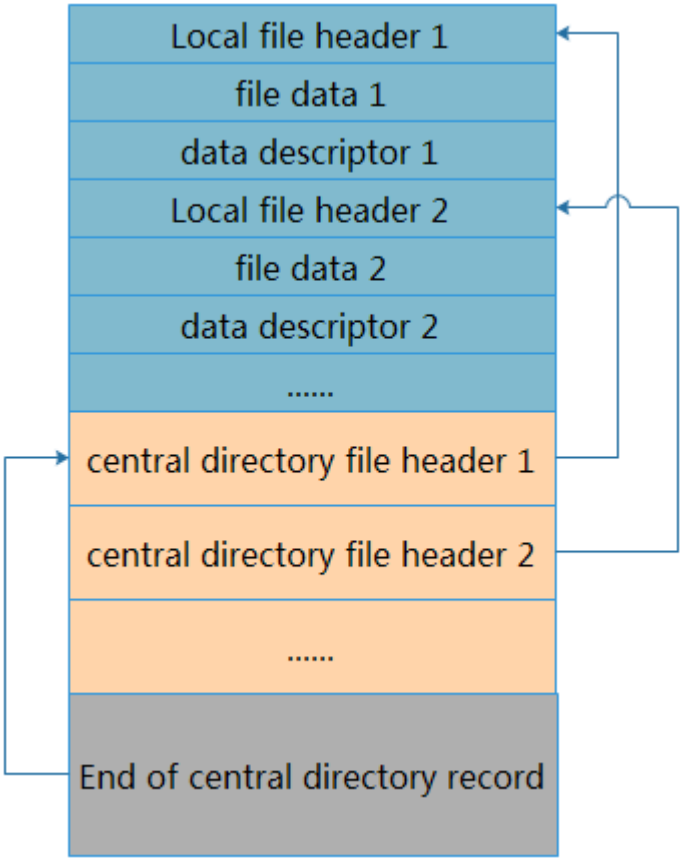
OK，了解了V1的签名和校验流程。我们来看下，V1签名是怎么保证APK文件不被篡改的？ 首先，如果破坏者修改了APK中的任何文件，那么被篡改文件的数据摘要的Base64编码就和 `MANIFEST.MF` 文件的记录值不一致，导致校验失败。 其次，如果破坏者同时修改了对应文件在 `MANIFEST.MF` 文件中的Base64值，那么MANIFEST.MF中对应数据块的Base64值就和 `CERT.SF` 文件中的记录值不一致，导致校验失败。 最后，如果破坏者更进一步，同时修改了对应文件在 `CERT.SF` 文件中的Base64值，那么 `CERT.SF` 的数字签名就和 `CERT.RSA` 记录的签名不一致，也会校验失败。 那有没有可能继续伪造 `CERT.SF` 的数字签名那？理论上不可能，因为破坏者没有开发者的私钥。那破坏者是不是可以用自己的私钥和数字证书重新签名那，这倒是完全可以！

综上所述，任何对APK文件的修改，在安装时都会失败，除非对APK重新签名。但是相同包名，不同签名的APK也是不能同时安装的。

APK文件结构

由上述V1签名和校验机制可知，修改APK中的任何文件都会导致安装失败！那怎么添加渠道信息那？只能从APK的结构入手了。

APK文件本质上是一个ZIP压缩包，而ZIP格式是固定的，主要由三部分构成，如下图所示：



第一部分是内容块，所有的压缩文件都在这部分。每个压缩文件都有一个`local file header`，主要记录了文件名、压缩算法、压缩前后的文件大小、修改时间、CRC32值等。 第二部分称为中央目录，包含了多个`central directory file header`（和第一部分的`local file header`一一对应），每个中央目录文件头主要记录了压缩算法、注释信息、对应`local file header`的偏移量等，方便快速定位数据。 最后一部分是EOCD，主要记录了中央目录大小、偏移量和ZIP注释信息等，其详细结构如下图所示：

End of central directory record (EOCD)

Offset	Bytes	Description[25]
0	4	End of central directory signature = 0x06054b50
4	2	Number of this disk
6	2	Disk where central directory starts
8	2	Number of central directory records on this disk
10	2	Total number of central directory records
12	4	Size of central directory (bytes)
16	4	Offset of start of central directory, relative to start of archive
20	2	Comment length (*n*)
22	*n*	Comment

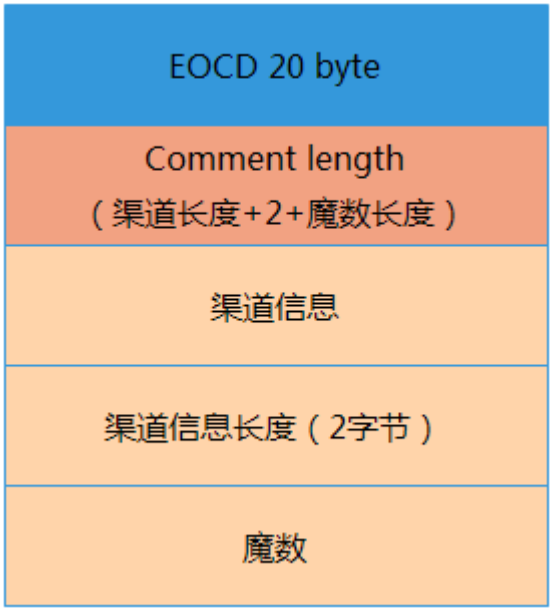
根据之前的V1签名和校验机制可知，V1签名只会检验第一部分的所有压缩文件，而不理会后两部分内容。因此，只要把渠道信息写入到后两块内容就可以通过V1校验，而EOCD的注释字段无疑是最好的选择。

基于V1签名的多渠道打包方案

既然找到了突破口，那么基于V1签名的多渠道打包方案就应运而生：***在APK文件的注释字段，添加渠道信息。*** 整个方案包括以下几步：

1. 复制APK
2. 找到EOCD数据块
3. 修改注释长度
4. 添加渠道信息
5. 添加渠道信息长度
6. 添加魔数

添加渠道信息后的EOCD数据块如下所示：



这里添加魔数的好处是方便从后向前读取数据，定位渠道信息。 因此，读取渠道信息包括以下几步：

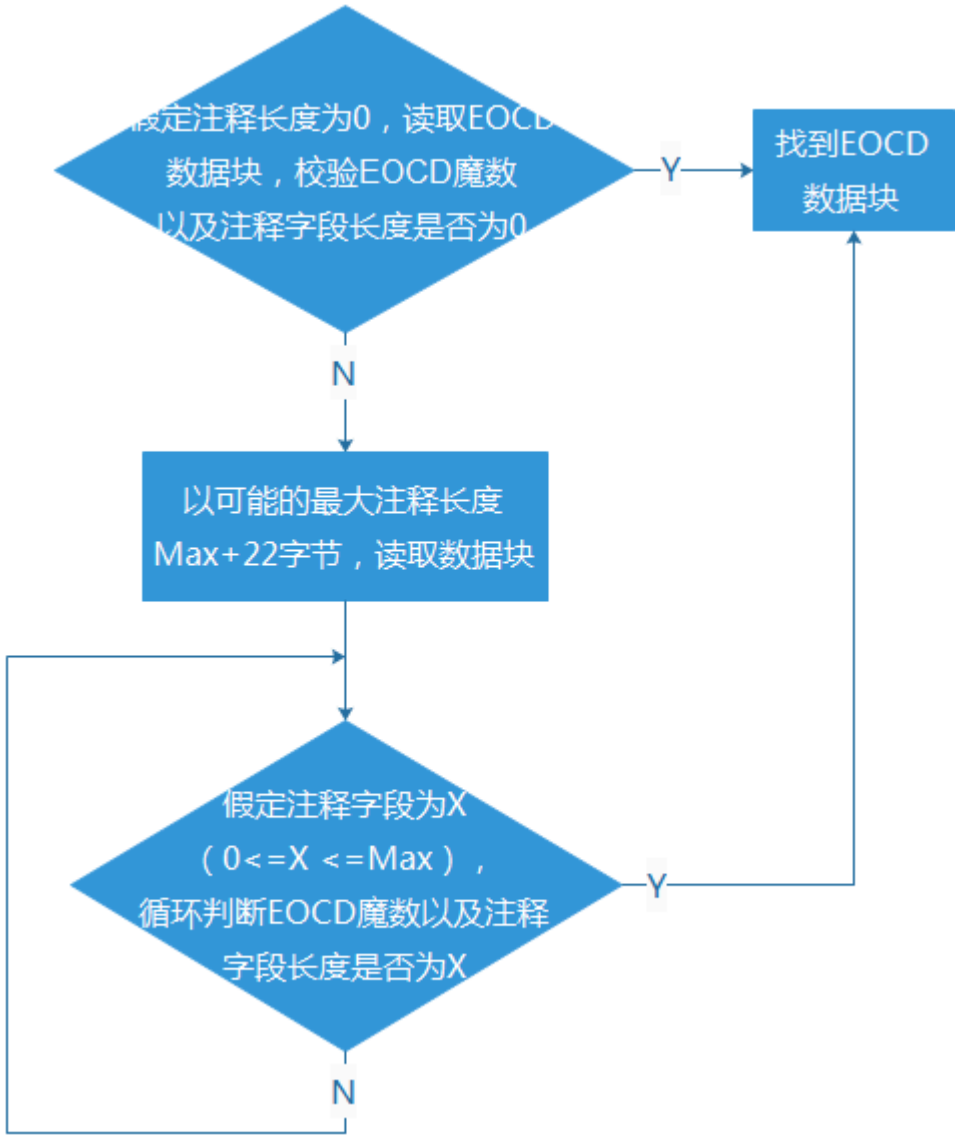
1. 定位到魔数
2. 向前读两个字节，确定渠道信息的长度LEN
3. 继续向前读LEN字节，就是渠道信息了。

通过16进制编辑器，可以查看到添加渠道信息后的APK（小端模式），如下所示：

```
16:8000h: 63 50 4B 05 06 00 00 00 00 93 01 93 01 43 99 00  cPK.....".".C™.
16:8010h: 00 BE E6 15 00 0E 00 6C 65 6F 6E 04 00 6C 74 6C  .¾æ....leon..ltl
16:8020h: 6F 76 65 7A 68                                   ovezh
```

`6C 74 6C 6F 76 75 7A 68`是魔数，`04 00`表示渠道信息长度为4，`6C 65 6F 6E`就是渠道信息`leon`了。`0E 00`就是APK注释长度了，正好是14。

虽说整个方案很清晰，但是在`找到EOCD数据块`这步遇到一个问题。如果APK本身没有注释，那最后22字节就是EOCD。但是若APK本身已经包含了注释字段，那怎么确定EOCD的起始位置那？这里借鉴了系统V2签名确定EOCD位置的方案。整个计算流程如下图所示：



整个方案介绍完了，该方案的最大优点就是：不需要解压缩APK，不需要重新签名，只需要复制APK，在注释字段添加渠道信息。每个渠道包仅需几秒的耗时，非常适合渠道较多的APK。

但是好景不长，Android7.0之后新增了V2签名，该签名会校验整个APK的数据摘要，导致上述渠道打包方案失效。所以如果想继续使用上述方案，需要关闭Gradle Plugin中的V2签名选项，禁用V2签名。

V2签名和多渠道打包方案

为什么需要V2签名

从前面的V1签名介绍，可以知道V1存在两个弊端：

1. `MANIFEST.MF`中的数据摘要是基于原始未压缩文件计算的。因此在校验时，需要先解压出原始文件，才能进行校验。而解压操作无疑是耗时的。
2. V1签名仅仅校验APK第一部分中的文件，缺少对APK的完整性校验。因此，在签名后，我们还可以修改APK文件，例如：通过zipalign进行字节对齐后，仍然可以正常安装。

正是基于这两点，Google提出了V2签名，解决了上述两个问题：

1. V2签名是对APK本身进行数据摘要计算，不存在解压APK的操作，减少了校验时间。
2. V2签名是针对整个APK进行校验（不包含签名块本身），因此对APK的任何修改（包括添加注释、zipalign字节对齐）都无法通过V2签名的校验。

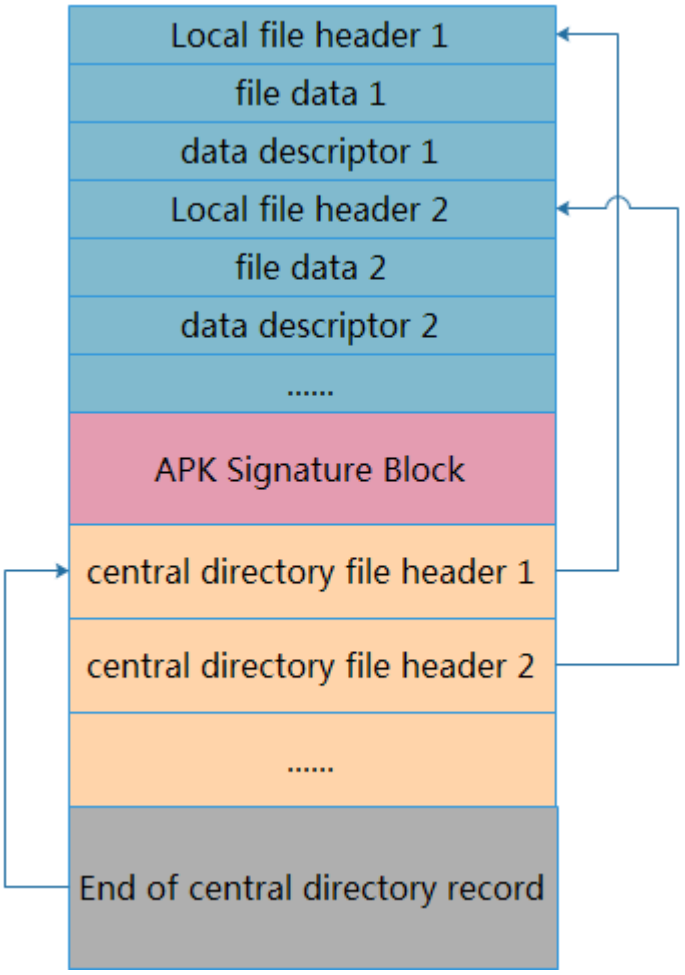
关于第一点的耗时问题，这里有一份实验室数据（Nexus 6P、Android 7.1.1）可供参考。

APK安装耗时对比	取5次平均耗时（秒）
V1签名APK	11.64
V2签名APK	4.42

可见，V2签名对APK的安装速度还是提升不少的。

V2签名机制

不同于V1，V2签名会生成一个签名块，插入到APK中。因此，V2签名后的APK结构如下图所示：



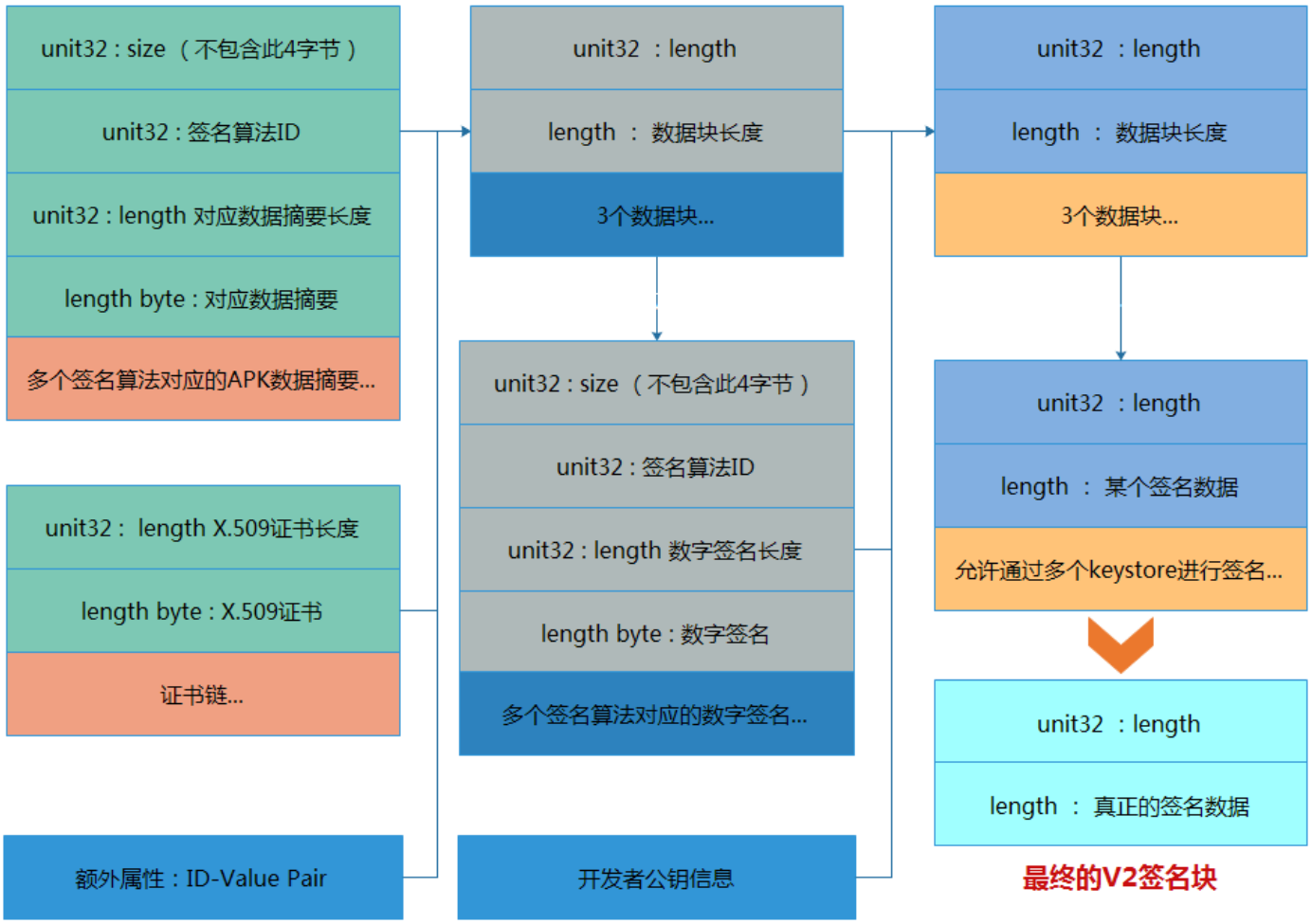
APK签名块位于中央目录之前，文件数据之后。***V2签名同时修改了EOCD中的中央目录的偏移量，使签名后的APK还符合ZIP结构。***

APK签名块的具体结构如下图所示：

 APK签名块

首先是8字节的签名块大小，此大小不包含该字段本身的8字节；其次就是ID-Value序列，就是一个4字节的ID和对应的数据；然后又是一个8字节的签名块大小，与开始的8字节是相等的；最后是16字节的签名块魔数。 其中，***ID为`0x7109871a`对应的Value就是V2签名块数据。***

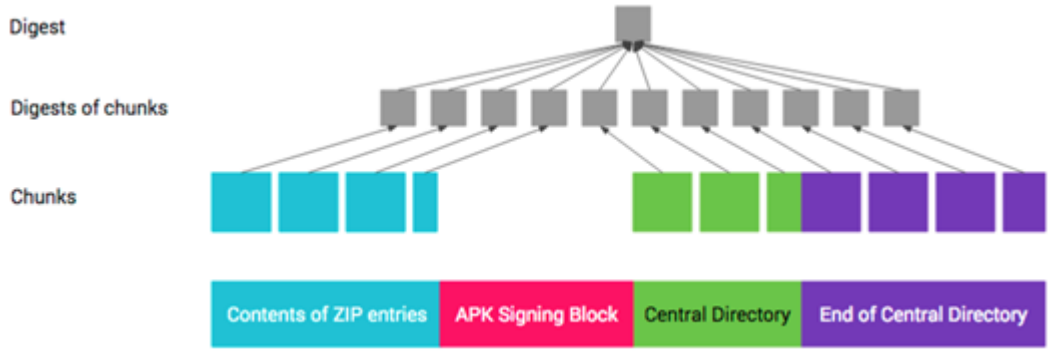
V2签名块的生成可参考ApkSignerV2，整体结构和流程如下图所示：



首先，根据多个签名算法，计算出整个APK的数据摘要，组成左上角的**APK数据摘要集**； 接着，把最左侧一列的`数据摘要`、`数字证书`和`额外属性`组装起来，形成类似于V1签名的“MF”文件（第二列第一行）； 其次，再用相同的私钥，不同的签名算法，计算出“MF”文件的数字签名，形成类似于V1签名的“SF”文件（第二列第二行）； 然后，把第二列的`类似MF文件`、`类似SF文件`和`开发者公钥`一起组装成通过单个keystore签名后的v2签名块（第三列第一行）。 最后，把多个keystore签名后的签名块组装起来，就是完整的V2签名块了（Android中允许使用多个keystore对apk进行签名）。

上述流程比较繁琐。简而言之，单个keystore签名块主要由三部分组成，分别是上图中第二列的三个数据块：`类似MF文件`、`类似SF文件`和`开发者公钥`，其结构如下图所示：

除此之外，Google也优化了计算数据摘要的算法，使得可以并行计算，如下图所示：



数据摘要的计算包括以下几步：

1. 首先，将上述APK中文件内容块、中央目录、EOCD按照1MB大小分割成一些小块。
2. 然后，计算每个小块的数据摘要，基础数据是0xa5 + 块字节长度 + 块内容。
3. 最后，计算整体的数据摘要，基础数据是0x5a + 数据块的数量 + 每个数据块的摘要内容。

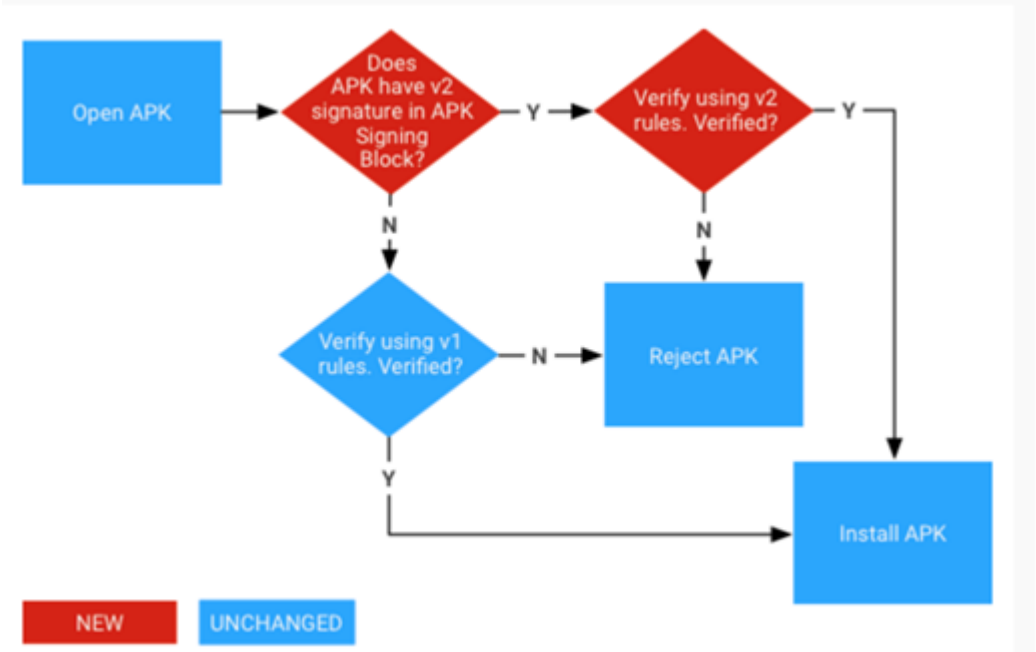
这样，每个数据块的数据摘要就可以并行计算，加快了V2签名和校验的速度。

V2校验流程

Android Gradle Plugin2.2之上默认会同时开启V1和V2签名，同时包含V1和V2签名的`CERT.SF`文件会有一个特殊的主属性，如下图所示：

```
Signature-Version: 1.0
X-Android-APK-Signed: 2
SHA-256-Digest-Manifest: 1bNAZl1AYvFQPb4ph7JsOZdhuE9cE4JXvfRA7z1E6BE=
Created-By: 1.0 (Android)
```

该属性会强制APK走V2校验流程（7.0之上），以充分利用V2签名的优势（速度快和更完善的校验机制）。 因此，同时包含V1和V2签名的APK的校验流程如下所示：



简而言之：优先校验V2，没有或者不认识V2，则校验V1。

这里引申出另外一个问题：APK签名时，只有V2签名，没有V1签名行不行？ 经过尝试，这种情况是可以编译通过的，并且在Android 7.0之上也可以正确安装和运行。但是7.0之下，因为不认识V2，又没有V1签名，所以会报没有签名的错误。

OK，明确了Android平台对V1和V2签名的校验选择之后，我们来看下V2签名的具体校验流程（`PackageManagerService.java` -> `PackageParser.java` -> `ApkSignatureSchemeV2Verifier.java`），如下图所示：

其中，最强签名算法是根据该算法使用的数据摘要算法来对比产生的，比如：SHA512 > SHA256。 校验成功的定义是至少找到一个keystore对应的签名块，并且所有签名块都按照上述流程校验成功。

下面我们来看下V2签名是怎么保证APK不被篡改的？ 首先，如果破坏者修改了APK文件的任何部分（签名块本身除外），那么APK的数据摘要就和“MF”数据块中记录的数据摘要不一致，导致校验失败。 其次，如果破坏者同时修改了“MF”数据块中的数据摘要，那么“MF”数据块的数字签名就和“SF”数据块中记录的数字签名不一致，导致校验失败。 然后，如果破坏者使用自己的私钥去加密生成“SF”数据块，那么使用开发者的公钥去解密“SF”数据块中的数字签名就会失败； 最后，更进一步，若破坏者甚至替换了开发者公钥，那么使用数字证书中的公钥校验签名块中的公钥就会失败，这也正是数字证书的作用。

综上所述，任何对APK的修改，在安装时都会失败，除非对APK重新签名。但是相同包名，不同签名的APK也是不能同时安装的。

到这里，V2签名已经介绍完了。但是在最后一步"数据摘要校验"这里，隐藏了一个点，不知道有没有人发现？ 因为，我们V2签名块中的数据摘要是针对APK的文件内容块、中央目录和EOCD三块内容计算的。但是在写入签名块后，修改了EOCD中的中央目录偏移量，那么在进行V2签名校验时，理论上在"数据摘要校验"这步应该会校验失败啊！但是为什么V2签名可以校验通过那?

这个问题很重要，因为我们下面要介绍的**基于V2签名的多渠道打包方案**也会修改EOCD的中央目录偏移量。

其实也很简单，原来Android系统在校验APK的数据摘要时，首先会把EOCD的中央目录偏移量替换成签名块的偏移量，然后再计算数据摘要。而签名块的偏移量不就是v2签名之前的中央目录偏移量嘛！！！，因此，这样计算出的数据摘要就和"MF"数据块中的数据摘要完全一致了。具体代码逻辑，可参考ApkSignatureSchemeV2Verifier.java的416 ~ 420行。

基于V2签名的多渠道打包方案

在上节V2签名的校验流程中，有一个很重要的细节：**Android系统只会关注ID为0x7109871a的V2签名块，并且忽略其他的ID-Value，同时V2签名只会保护APK本身，不包含签名块。**

因此，基于V2签名的多渠道打包方案就应运而生：***在APK签名块中添加一个ID-Value，存储渠道信息。*** 整个方案包括以下几步：

1. 找到APK的EOCD块
2. 找到APK签名块
3. 获取已有的ID-Value Pair
4. 添加包含渠道信息的ID-Value
5. 基于所有的ID-Value生成新的签名块
6. 修改EOCD的中央目录的偏移量（上面已介绍过：修改EOCD的中央目录偏移量，不会导致数据摘要校验失败）
7. 用新的签名块替代旧的签名块，生成带有渠道信息的APK

实际上，除了渠道信息，我们可以在APK签名块中添加任何辅助信息。

通过16进制编辑器，可以查看到添加渠道信息后的APK（小端模式），如下所示：

```
15:87F0h: 3A DB CB D6 CE E4 B5 02 03 01 00 01 08 00 00 00  :ÛËÖÎäµ.........
15:8800h: 00 00 00 00 FF 55 11 88 6C 65 6F 6E 29 06 00 00  ....ÿU.ˆleon)...
15:8810h: 00 00 00 00 41 50 4B 20 53 69 67 20 42 6C 6F 63  ....APK Sig Bloc
```

`6C 65 6F 6E` 就是我们的渠道信息 `leon`。向前4个字节：`FF 55 11 88` 就是我们添加的ID，再向前8个字节：`08 00 00 00 00 00 00 00` 就是我们的ID-Value的长度，正好是8。

整个方案介绍完了，该方案的最大优点就是：支持7.0之上新增的V2签名，同时兼有V1方案的所有优点。

多渠道包的强校验

那么如何保证通过这些方案生成的渠道包，能够在所有Android平台上正确安装那?

原来Google提供了一个同时支持V1和V2签名和校验的工具：apksig。它包括一个`apksigner`命令行和一个`apksig`类库。其中前者就是Android SDK build-tools下面的命令行工具。而我们正是借助后面的apksig来进行渠道包强校验，它可以保证渠道包在apk Minsdk ~ 最高版本之间都校验通过。

多渠道打包工具对比

目前市面上的多渠道打包工具主要有packer-ng-plugin和美团的Walle。下表是我们的VasDolly和它们之间的简单对比。

多渠道打包工具对比	VasDolly	packer-ng-plugin	Walle
V1签名方案	支持	支持	不支持
V2签名方案	支持	不支持	支持
已有注释块的APK	支持	不支持	不支持
根据已有APK生成渠道包	支持	不支持	不支持
命令行工具	支持	支持	支持
强校验	支持	不支持	不支持
多线程加速打包	支持	不支持	不支持

这里我之所以同时支持V1和V2签名方案，主要是担心后续Android平台加强签名校验机制，导致V2多渠道打包方案行不通，可以无痛切换到V1签名方案。

Clone this wiki locally

```
https://github.com/Tencent/VasDolly.wiki.git
```