

iOS App 签名的原理

Source: <http://blog.cnbang.net/tech/3386/>

iOS签名机制挺复杂，各种证书，Provisioning Profile，Entitlements，CertificateSigningRequest，p12，AppID，概念一堆，也很容易出错，本文尝试从原理出发，一步步推出为什么会有这么多概念，希望能有助于理解iOS App签名的原理和流程。

目的

先来看看苹果的签名机制是为了做什么。在iOS出来之前，在主流操作系统(Mac/Windows/Linux)上开发和运行软件是不需要签名的，软件随便从哪里下载都能运行，导致平台对第三方软件难以控制，盗版流行。苹果希望解决这样的问题，在iOS平台对第三方APP有绝对的控制权，一定要保证每一个安装到iOS上的APP都是经过苹果官方允许的，怎样保证呢？就是通过签名机制。

非对称加密

通常我们说的签名就是数字签名，它是基于非对称加密算法实现的。对称加密是通过同一份密钥加密和解密数据，而非对称加密则有两份密钥，分别是公钥和私钥，用公钥加密的数据，要用私钥才能解密，用私钥加密的数据，要用公钥才能解密。

简单说一下常用的非对称加密算法RSA的数学原理，理解简单的数学原理，就可以理解非对称加密是怎么做到的，为什么会是安全的：

1. 选两个质数 p 和 q ，相乘得出一个大整数 n ，例如 $p = 61$ ， $q = 53$ ， $n = pq = 3233$ ；
2. 选 $1-n$ 间的随便一个质数 e ，例如 $e = 17$ ；
3. 经过一系列数学公式，算出一个数字 d ，满足：**a.** 通过 n 和 e 这两个数据一组数据进行数学运算后，可以通过 n 和 d 去反解运算，反过来也可以。**b.** 如果只知道 n 和 e ，要推导出 d ，需要知道 p 和 q ，也就是需要把 n 因数分解。

上述的 (n,e) 这两个数据在一起就是公钥， (n,d) 这两个数据就是私钥，满足用私钥加密，公钥解密，或反过来公钥加密，私钥解密，也满足在只暴露公钥(只知道 n 和 e)的情况下，要推导出私钥 (n,d) ，需要把大整数 n 因数分解。目前因数分解只能靠暴力穷举，而 n 数字越大，越难以用穷举计算出因数 p 和 q ，也就越安全，当 n 大到二进制1024位或2048位时，以目前技术要破解几乎不可能，所以非常安全。

若对数字 d 是怎样计算出来的感兴趣，可以详读这两篇文章：RSA 算法原理：

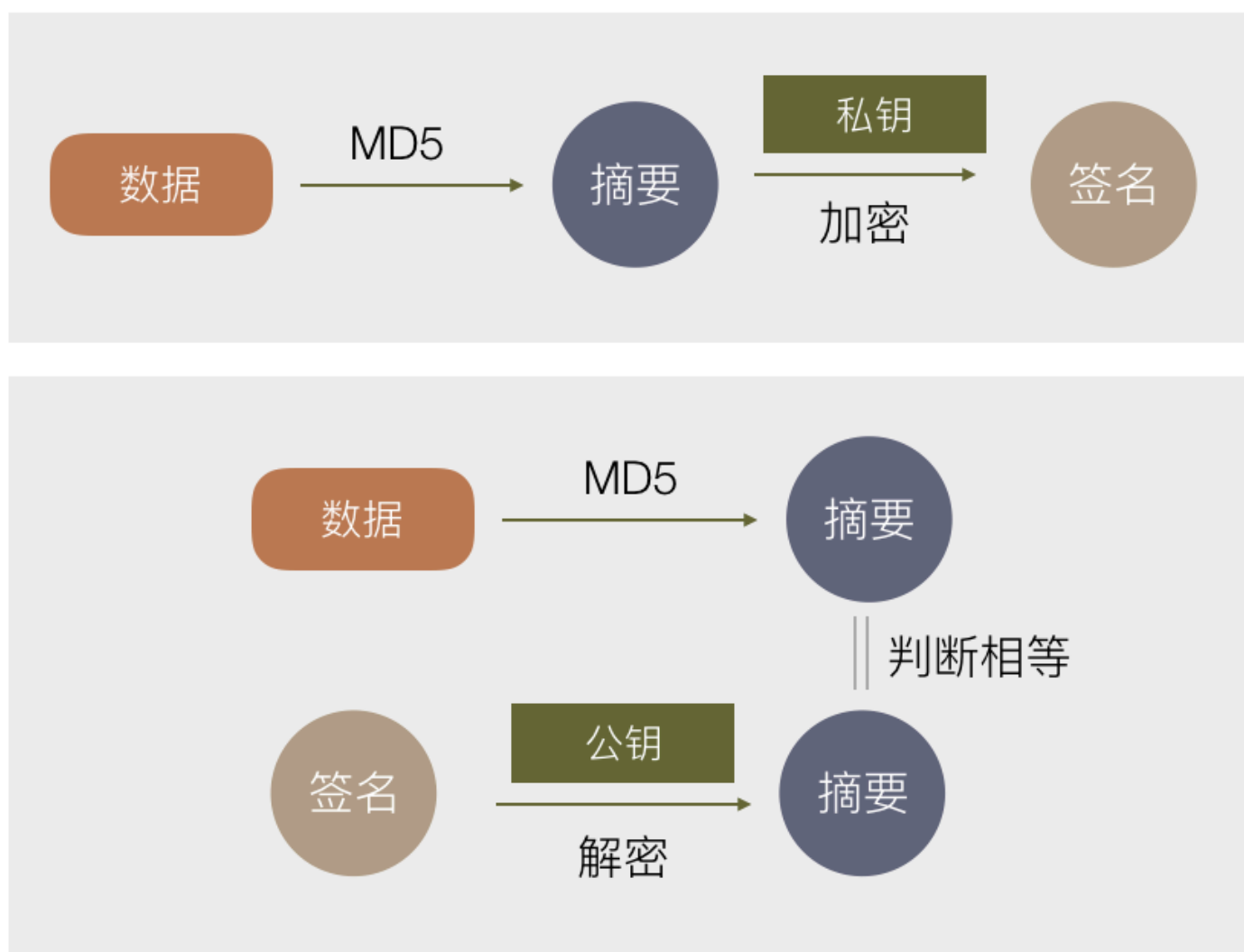
- * [RSA算法原理（一）](#)
- * [RSA算法原理（二）](#)

数字签名

现在知道了有非对称加密这东西，那数字签名是怎么回事呢？

数字签名的作用是我对某一份数据打个标记，表示我认可了这份数据（签了个名），然后我发送给其他人，其他人可以知道这份数据是经过我认证的，数据没有被篡改过。

有了上述非对称加密算法，就可以实现这个需求：



1. 首先用一种算法，算出原始数据的摘要。需满足：**a.** 若原始数据有任何变化，计算出来的摘要值都会变化。**b.** 摘要要够短。这里最常用的算法是MD5。
2. 生成一份非对称加密的公钥和私钥，私钥我自己拿着，公钥公布出去。
3. 对一份数据，算出摘要后，用私钥加密这个摘要，得到一份加密后的数据，称为原始数据的签名。把它跟原始数据一起发送给用户。

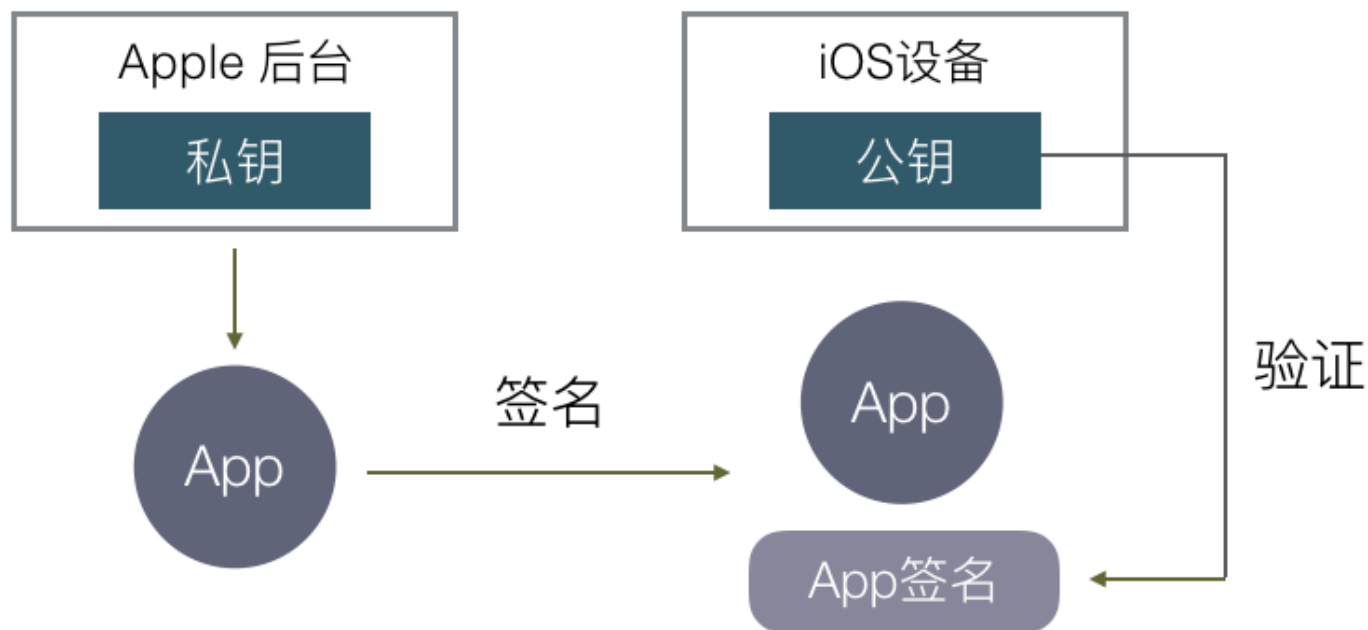
4. 用户收到数据和签名后，用公钥解密得到摘要。同时用户用同样的算法计算原始数据的摘要，对比这里计算出来的摘要和用公钥解密签名得到的摘要是否相等，若相等则表示这份数据中途没有被篡改过，因为如果篡改过，摘要会变化。

之所以要有第一步计算摘要，是因为非对称加密的原理限制可加密的内容不能太大（不能大于上述 n 的位数，也就是一般不能大于 1024 位/ 2048 位），于是若要对任意大的数据签名，就需要改成对它的特征值签名，效果是一样的。

好了，有了非对称加密的基础，知道了数字签名是什么，怎样可以保证一份数据是经过某个地方认证的，来看看怎样通过数字签名的机制保证每一个安装到iOS上的APP都是经过苹果认证允许的。

最简单的签名

要实现这个需求很简单，最直接的方式，苹果官方生成一对公私钥，在iOS里内置一个公钥，私钥由苹果后台保存，我们传App上App Store时，苹果后台用私钥对APP数据进行签名，iOS系统下载这个APP后，用公钥验证这个签名，若签名正确，这个APP肯定是由苹果后台认证的，并且没有被修改过，也就达到了苹果的需求：保证安装的每一个APP都是经过苹果官方允许的。



如果我们iOS设备安装APP只有从App Store下载这一种方式的话，这件事就结束了，没有任何复杂的东西，只有一个数字签名，非常简单地解决问题。

但实际上因为除了从App Store下载，我们还可以有三种方式安装一个App：

1. 开发App时可以直接把开发中的应用安装进手机进行调试。

2. In-House企业内部分发，可以直接安装企业证书签名后的APP。
3. AD-Hoc相当于企业分发的限制版，限制安装设备数量，较少用。

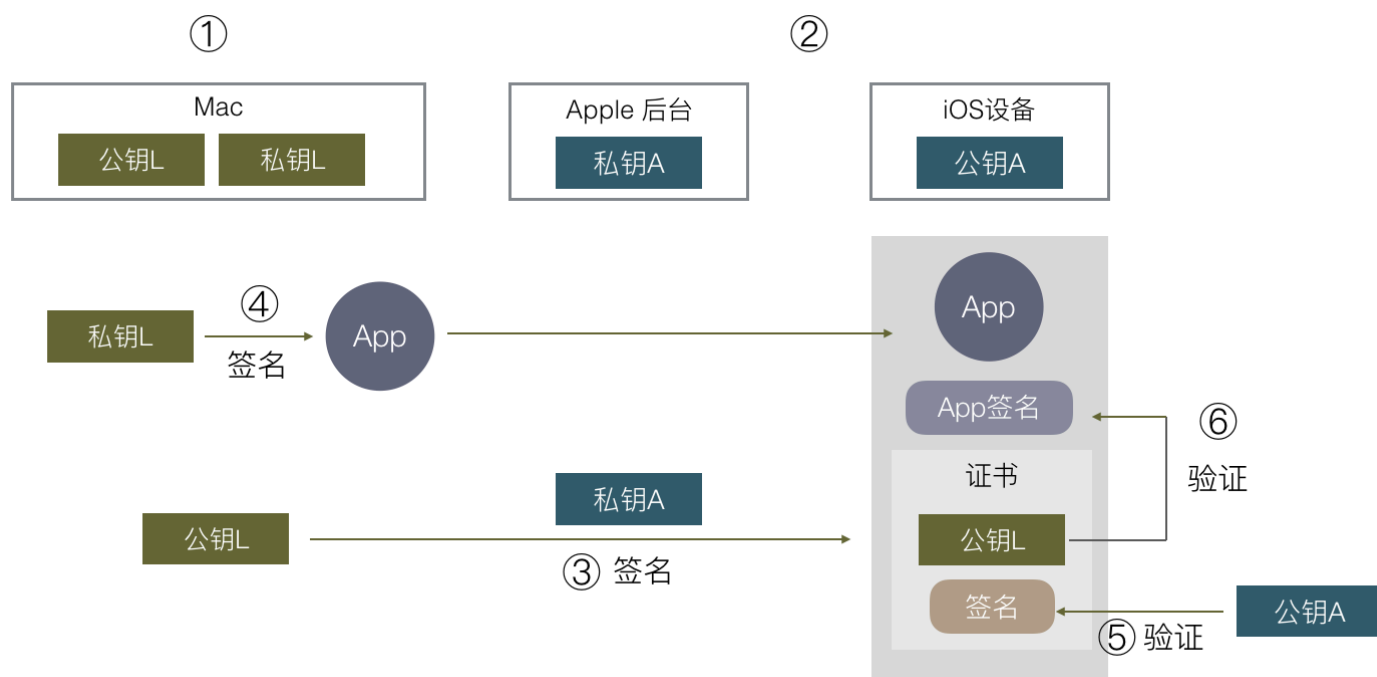
苹果要对用这三种方式安装的App进行控制，就有了新的需求，无法像上面这样简单了。

新的需求

我们先来看第一个，开发时安装APP，它有两个需求：

1. 安装包不需要传到苹果服务器，可以直接安装到手机上。如果你编译一个APP到手机前要先传到苹果服务器签名，这显然是不能接受的。
2. 苹果必须对这里的安装有控制权，包括：**a.** 经过苹果允许才可以这样安装。**b.** 不能被滥用导致非开发App也能被安装。

为了实现这些需求，iOS签名的复杂度也就开始增加了。苹果这里给出的方案是使用了双层签名，会比较绕，流程大概是这样的：



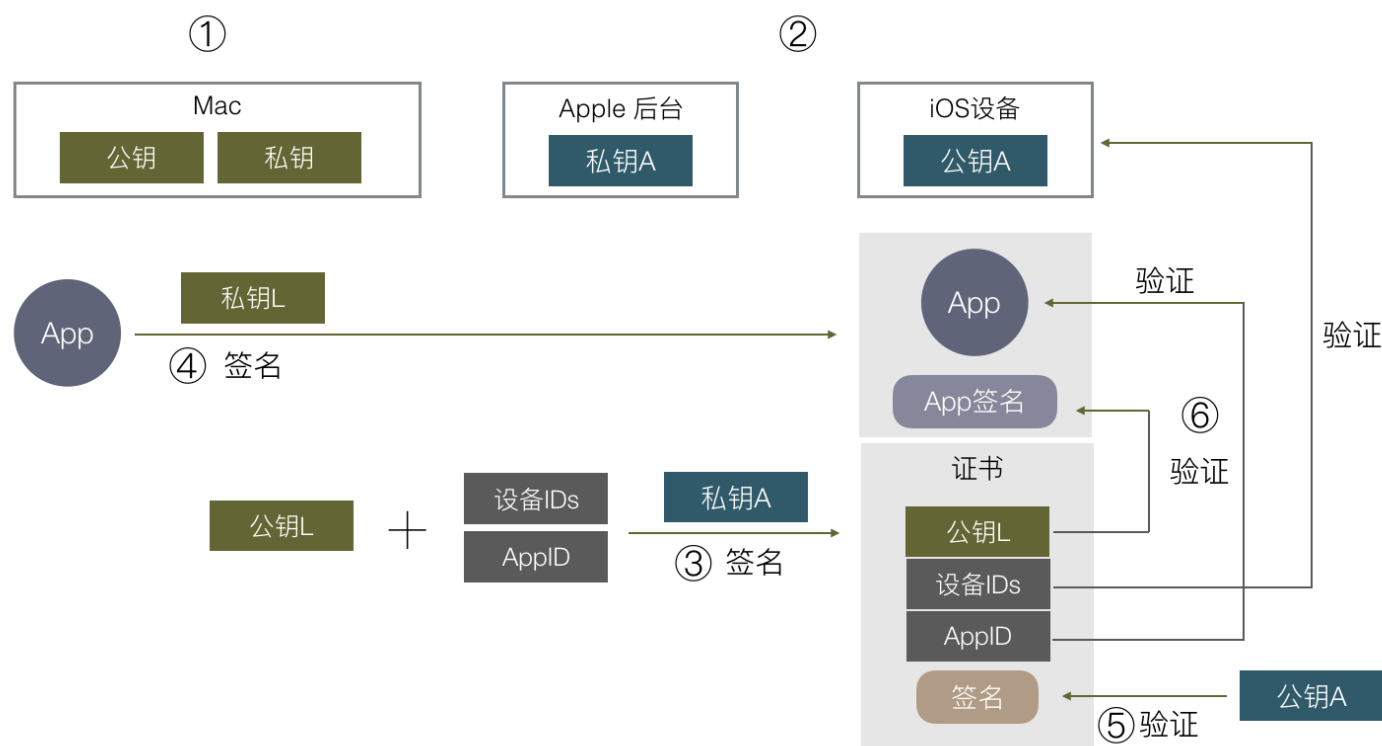
1. 在你的Mac开发机器生成一对公私钥，这里称为 **公钥L**，**私钥L**；L:Local。
2. 苹果自己有固定的一对公私钥，跟上面App Store例子一样，私钥在苹果后台，公钥在每个iOS设备上。这里称为 **公钥A**，**私钥A**；A:Apple。
3. 把 **公钥L** 传到苹果后台，用苹果后台里的 **私钥A** 去签名 **公钥L**。得到一份数据包含了 **公钥L** 以及其签名，把这份数据称为**证书**。
4. 在开发时，编译完一个APP后，用本地的 **私钥L** 对这个APP进行签名，同时把第三步得到的证书一起打包进APP里，安装到手机上。
5. 在安装时，iOS系统取得证书，通过系统内置的 **公钥A**，去验证证书的数字签名是否正确。

6. 验证证书后确保了 公钥L 是苹果认证过的，再用 公钥L 去验证APP的签名，这里就间接验证了这个APP安装行为是否经过苹果官方允许。（这里只验证安装行为，不验证APP是否被改动，因为开发阶段APP内容总是不断变化的，苹果不需要管。）

加点东西

上述流程只解决了上面第一个需求，也就是需要经过苹果允许才可以安装，还未解决第二个避免被滥用的问题。怎么解决呢？苹果再加了两个限制，一是限制在苹果后台注册过的设备才可以安装，二是限制签名只能针对某一个具体的APP。

怎么加的？在上述第三步，苹果用 私钥A 签名我们本地 公钥L 时，实际上除了签名 公钥L ，还可以加上无限多数据，这些数据都可以保证是经过苹果官方认证的，不会有被篡改的可能。



可以想到把允许安装的设备ID列表和App对应的AppID等数据，都在第三步这里跟 公钥L 一起组成证书，再用苹果 私钥A 对这个证书签名。在最后第5步验证时就可以拿到设备ID列表，判断当前设备是否符合要求。根据数字签名的原理，只要数字签名通过验证，第5步这里的设备IDs/AppID/ 公钥L 就都是经过苹果认证的，无法被修改，苹果就可以限制可安装的设备 and APP，避免滥用。

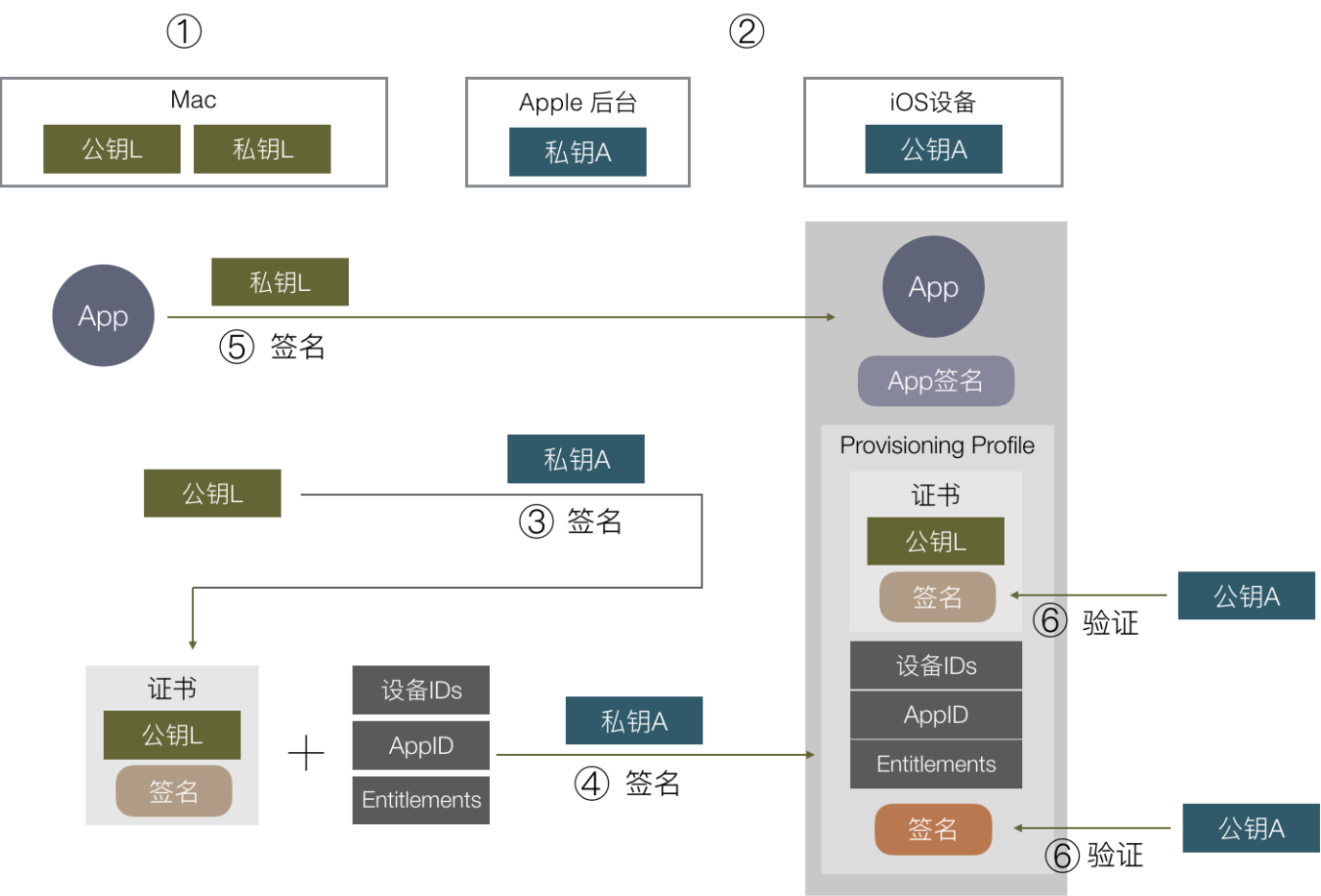
最终流程

到这里这个证书已经变得很复杂了，有很多额外信息，实际上除了设备ID/AppID，还有其他信息

也需要在这里用苹果签名，像这个APP里iCloud/Push/后台运行等权限苹果都想控制，苹果把这些权限开关统一称为Entitlements，它也需要通过签名去授权。

实际上一个“证书”本来就有规定的格式规范，上面我们把各种额外信息塞入证书里是不合适的，于是苹果另外搞了个东西，叫Provisioning Profile，一个Provisioning Profile里就包含了证书以及上述提到的所有额外信息，以及所有信息的签名。

所以整个流程稍微变一下，就变成这样了：



因为步骤有小变动，这里我们重新再列一遍整个流程：

1. 在你的Mac开发机器生成一对公私钥，这里称为 公钥L ， 私钥L ； L:Local。
2. 苹果自己有固定的一对公私钥，跟上面App Store例子一样，私钥在苹果后台，公钥在每个iOS设备上。这里称为 公钥A ， 私钥A ； A:Apple。
3. 把 公钥L 传到苹果后台，用苹果后台里的 私钥A 去签名 公钥L 。得到一份数据包含了 公钥L 以及其签名，把这份数据称为证书。
4. 在苹果后台申请AppID，配置好设备ID列表和APP可使用的权限，再加上第③步的证书，组成的数据用 私钥A 签名，把数据和签名一起组成一个Provisioning Profile文件，下载到本地Mac开发机。
5. 在开发时，编译完一个APP后，用本地的 私钥L 对这个APP进行签名，同时把第④步得到的Provisioning Profile文件打包进APP里，文件名为embedded.mobileprovision，把APP安装

到手机上。

6. 在安装时，iOS系统取得证书，通过系统内置的 公钥A，去验证embedded.mobileprovision的数字签名是否正确，里面的证书签名也会再验一遍。
7. 确保了embedded.mobileprovision里的数据都是苹果授权以后，就可以取出里面的数据，做各种验证，包括用 公钥L 验证APP签名，验证设备ID是否在ID列表上，AppID是否对应得上，权限开关是否跟APP里的Entitlements对应等。

开发者证书从签名到认证最终苹果采用的流程大致是这样，还有一些细节像证书有效期/证书类型等就不细说了。

概念和操作

上面的步骤对应到我们平常具体的操作和概念是这样的：

1. 第1步对应的是keychain里的“从证书颁发机构请求证书”，这里就本地生成了一对公私钥，保存的CertificateSigningRequest就是公钥，私钥保存在本地电脑里。
2. 第2步苹果处理，不用管。
3. 第3步对应把CertificateSigningRequest传到苹果后台生成证书，并下载到本地。这时本地有两个证书，一个是第1步生成的，一个是这里下载回来的，keychain会把这两个证书关联起来，因为他们公私钥是对应的，在XCode选择下载回来的证书时，实际上会找到keychain里对应的私钥去签名。这里私钥只有生成它的这台Mac有，如果别的Mac也要编译签名这个App怎么办？答案是把私钥导出给其他Mac用，在keychain里导出私钥，就会存成 .p12 文件，其他Mac打开后就导入了这个私钥。
4. 第4步都是在苹果网站上操作，配置AppID/权限/设备等，最后下载Provisioning Profile文件。
5. 第5步XCode会通过第3步下载回来的证书（存着公钥），在本地找到对应的私钥（第一步生成的），用本地私钥去签名App，并把Provisioning Profile文件命名为embedded.mobileprovision一起打包进去。这里对App的签名数据保存分两部分，Mach-O可执行文件会把签名直接写入这个文件里，其他资源文件则会保存在_CodeSignature目录下。
6. 第6-7步的打包和验证都是Xcode和iOS系统自动做的事。

这里再总结一下这些概念：

1. 证书：内容是公钥或私钥，由其他机构对其签名组成的数据包。
2. Entitlements：包含了App权限开关列表。
3. CertificateSigningRequest：本地公钥。
4. p12：本地私钥，可以导入到其他电脑。
5. Provisioning Profile：包含了证书/Entitlements等数据，并由苹果后台私钥签名的数据包。

其他发布方式

前面以开发包为例子说了签名和验证的流程，另外两种方式In-House企业签名和AD-Hoc流程也是差不多的，只是企业签名不限制安装的设备数，另外需要用户在iOS系统设置上手动点击信任这个企业才能通过验证。

而App Store的签名验证方式有些不一样，前面我们说到最简单的签名方式，苹果在后台直接用私钥签名App就可以了，实际上苹果确实是这样做的，如果去下载一个App Store的安装包，会发现它里面是没有embedded.mobileprovision文件的，也就是它安装和启动的流程是不依赖这个文件，验证流程也就跟上述几种类型不一样了。

据猜测，因为上传到App Store的包苹果会重新对内容加密，原来的本地私钥签名就没有用了，需要重新签名，从App Store下载的包苹果也并不打算控制它的有效期，不需要内置一个embedded.mobileprovision去做校验，直接在苹果用后台的私钥重新签名，iOS安装时用本地公钥验证App签名就可以了。

那为什么发布App Store的包还是要跟开发版一样搞各种证书和Provisioning Profile？猜测因为苹果想做统一管理，Provisioning Profile里包含一些权限控制，AppID的检验等，苹果不想在上传App Store包时重新用另一种协议做一遍这些验证，就不如统一把这部分放在Provisioning Profile里，上传App Store时只要用同样的流程验证这个Provisioning Profile是否合法就可以了。

所以App上传到App Store后，就跟你的证书/Provisioning Profile都没有关系了，无论他们是否过期或被废除，都不会影响App Store上的安装包。

到这里iOS签名机制的原理和主流程大致说完了，希望能对理解苹果签名和排查日常签名问题有所帮助。

P.S. 一些疑问

最后这里再提一下我关于签名流程的一些疑问。

企业证书

企业证书签名因为限制少，在国内被广泛用于测试和盗版，fir.im/蒲公英等测试平台都是通过企业证书分发，国内一些市场像PP助手，爱思助手，一部分安装手段也是通过企业证书重签名。通过企业证书签名安装的App，启动时都会验证证书的有效期，并且不定期请求苹果服务器看证书是否被吊销，若已过期或被吊销，就会无法启动App。对于这种助手的盗版安装手段，苹果想打击只能一个个吊销企业证书，并没有太好的办法。

这里我的疑问是，苹果做了那么多签名和验证机制去限制在iOS安装App，为什么又要出这样一

个限制很少的方式让盗版钻空子呢？若真的是企业用途不适合上App Store，也完全可以在App Store开辟一个小的私密版块，还是通过App Store去安装，就不会有这个问题了。

App Store 加密

另一个问题是我们把App传上App Store后，苹果会对App进行加密，导致App体积增大不少，这个加密实际上是没卵用的，只是让破解的人要多做一个步骤，运行App去内存dump出可执行文件而已，无论怎样加密，都可以用这种方式拿出加密前的可执行文件。所以为什么要做这样的加密呢？想不到有什么好处。

本地私钥

我们看到前面说的签名流程很绕很复杂，经常出现各种问题，像有Provisioning Profile文件但证书又不对，本地有公钥证书没对应私钥等情况，不理解原理的情况下会被绕晕，我的疑问是，这里为什么不能简化呢？还是以开发证书为例，为什么一定要用本地Mac生成的私钥去签名？苹果要的只是本地签名，私钥不一定要本地生成的，苹果也可以自己生成一对公私钥给我们，放在Provisioning Profile里，我们用里面的私钥去加密就行了，这样就不会有CertificateSigningRequest和p12的概念，跟本地keychain没有关系，不需要关心证书，只要有Provisioning Profile就能签名，流程会减少，易用性会提高很多，同时苹果想要的控制一点都不会少，也没有什么安全问题，为什么不这样设计呢？

能想到的一个原因是Provisioning Profile在非App Store安装时会打包进安装包，第三方拿到这个Provisioning Profile文件就能直接用起来给他自己的App签名了。但这种问题也挺好解决，只需要打包时去掉文件里的私钥就行了，所以仍不明白为什么这样设计。