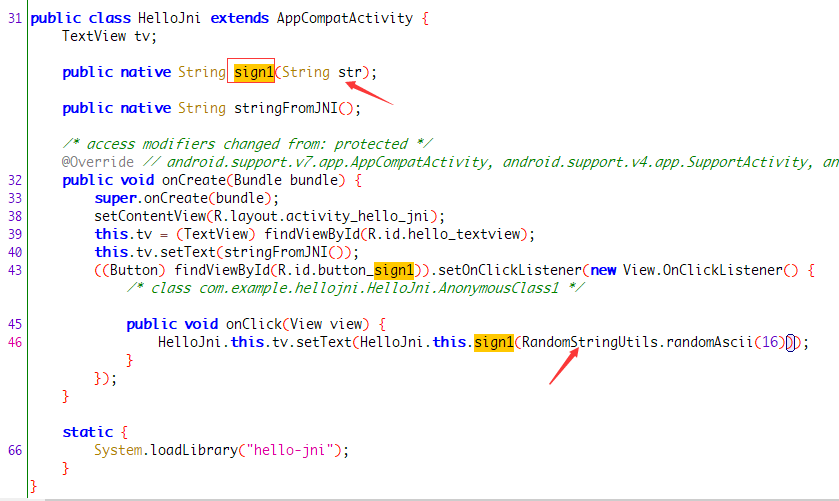
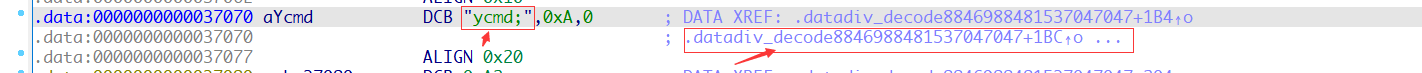
目标： 分析sign1算法

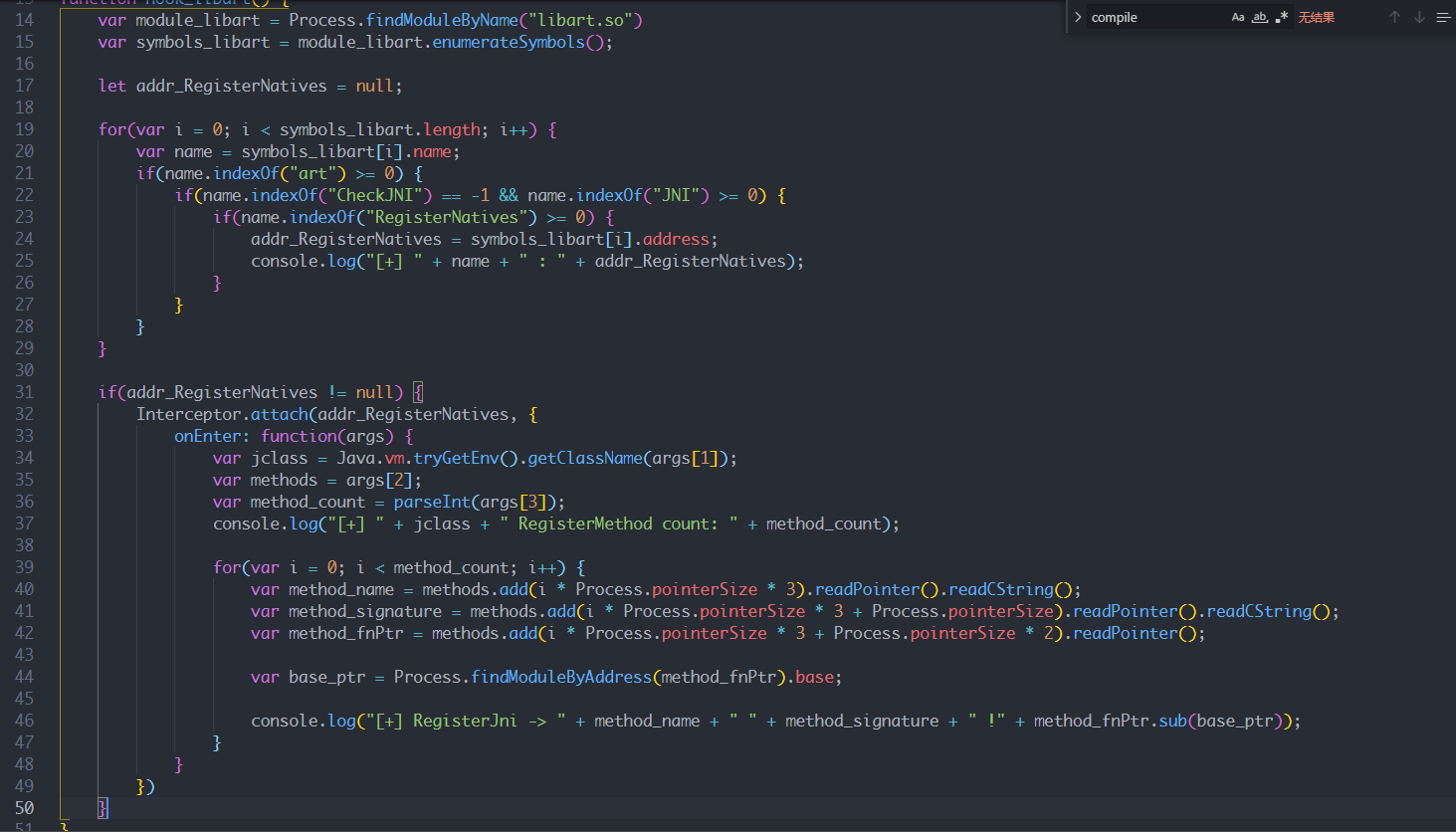
1. Java层发现调用native函数sign1



1. 进入后，发现没有静态注册函数，到JNI\_Onload找动态注册，发现RegisterNatives函数中注册的jni方法名不对，发现该data端被.init\_array中的函数调用。猜测是字符串被加密了，在程序启动时，通过.init\_array段进行解密。.datadiv\_decode是个明显的ollvm

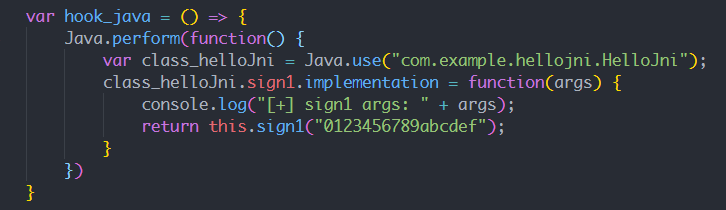


1. 通过hook RegisterNatives函数来获得函数名、函数签名和地址





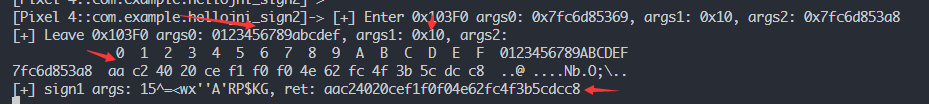
1. 由于java层输入是一个随机16位字符串，为了方便分析，我们把java层的输入，通过hook给固定住



1. 发现固定住之后点击还是会变化，但是实在三个值之间来回变化，所以应该是在so层还有一个随机数。进入0xe76c看so层sign1的具体实现
2. 观察代码，明显输出和sub\_103F0相关，先hook对应的函数参数和输出



1. Hook之后发现参数分别代表输入，长度，输出

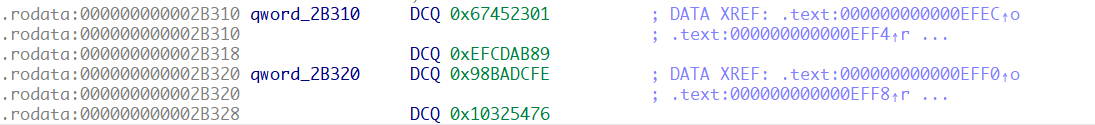


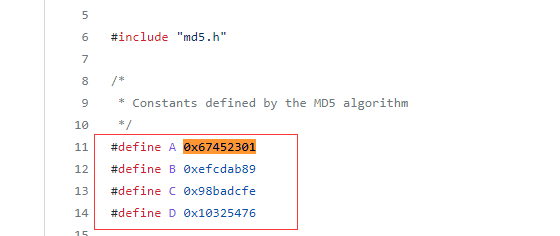
1. 查看sub\_103F0-》sub\_1005C

看到这两个常量习惯性的点进去看下

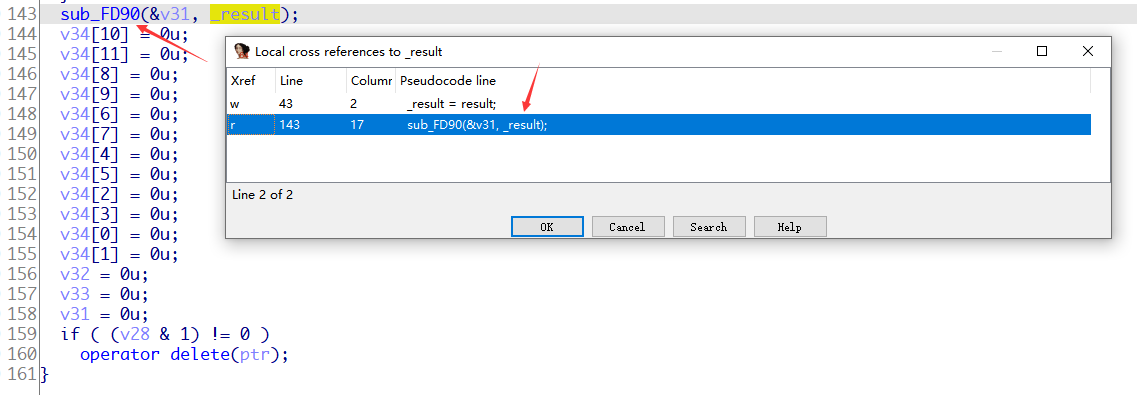


D了两下，像常量，习惯性的搜一搜，MD5的常量。奇怪的是这两个常量在代码中实际上都没有调用

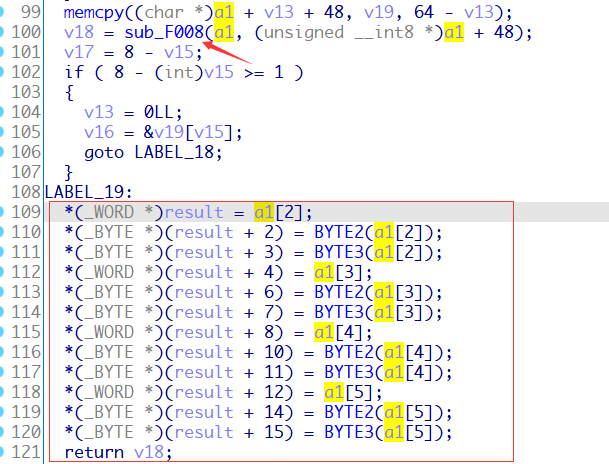




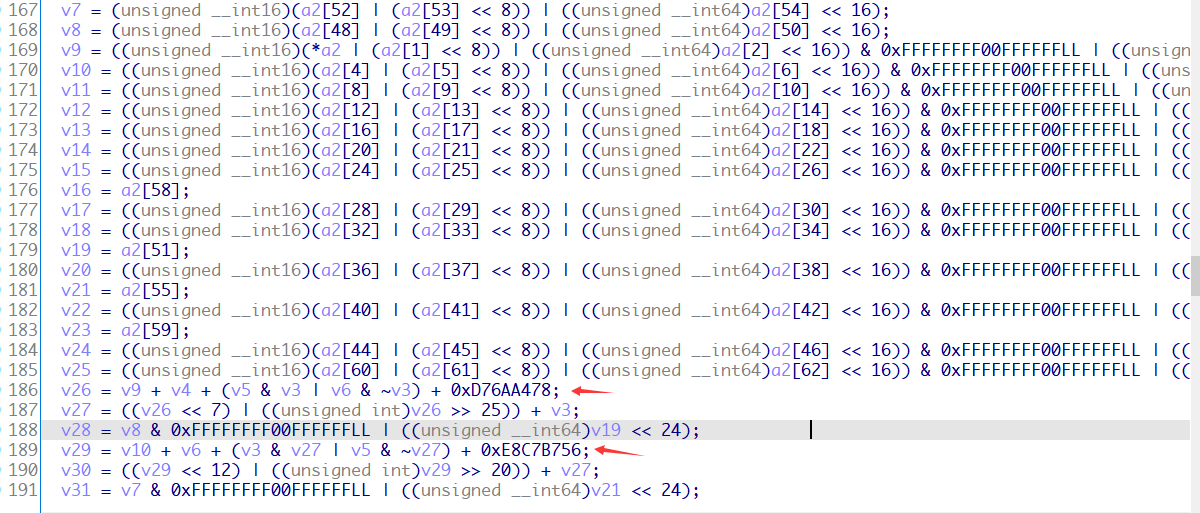
1. 从结果出发，发现结果主要与sub\_FD90相关



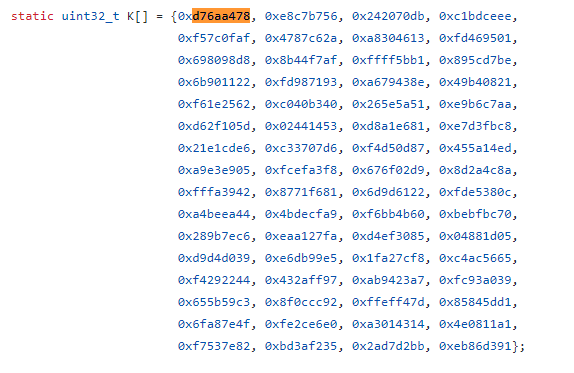
1. 从结果往上查，关键函数为sub\_F008



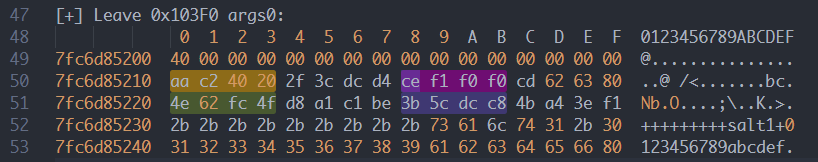
进去后发现大量移位运算，按H看下两个常量

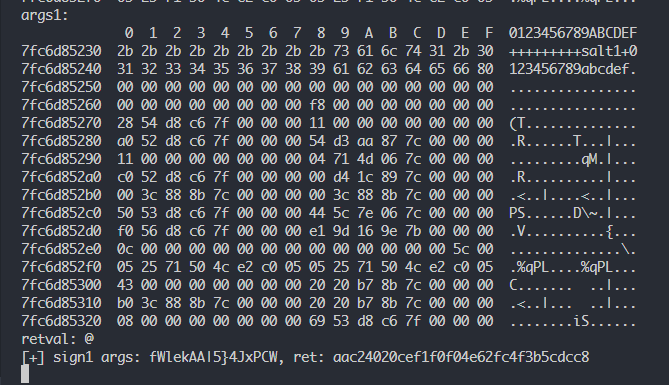


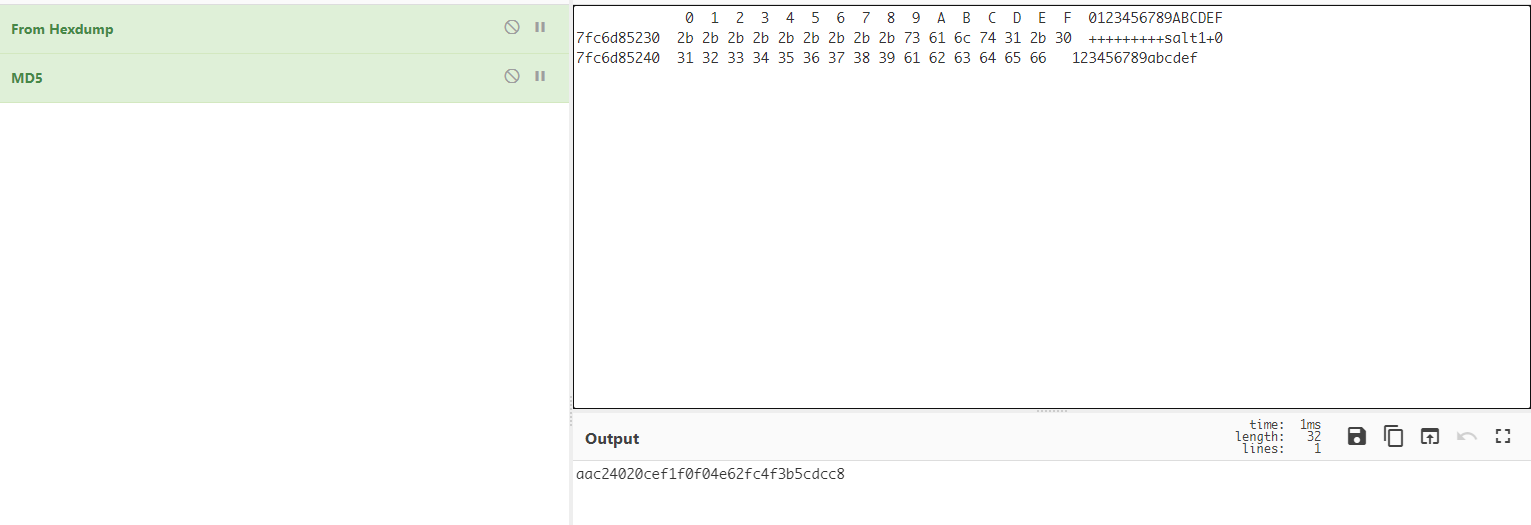
常量是MD5的常量表，猜测这里是MD5算法。



1. Hook值搞到MD5里面试一下，基本可以确定这个函数的作用是MD5





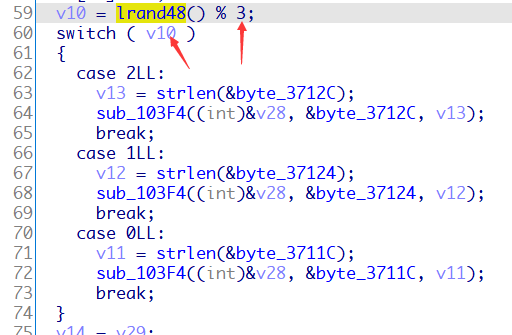


1. 现在来看MD5的输入是怎么来的，后半段明显是我们的输入。

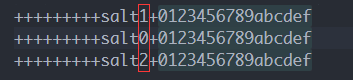


开始时说了，固定java层参数之后大概有3种不同的输出。

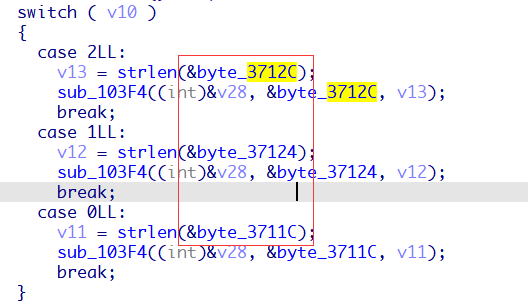
Sub\_1005c有个明显的随机然后mod 3的操作。



这里是直接用case的三种情况，我们直接多点几次遍历出来。



1. 这里data里的值都被ollvm在.init\_array段给混淆了，我们打印出来看看



分别是这三个，基本证明了我们的猜测

