由于像IDA等大量反汇编工具使用启发式的递归反汇编，所以通常使用间接寻址来对抗相应的静态分析。像阿里、美团、快手等，使用LR + 跳转表来实现间接跳转，并以CFF的结构进行组织。

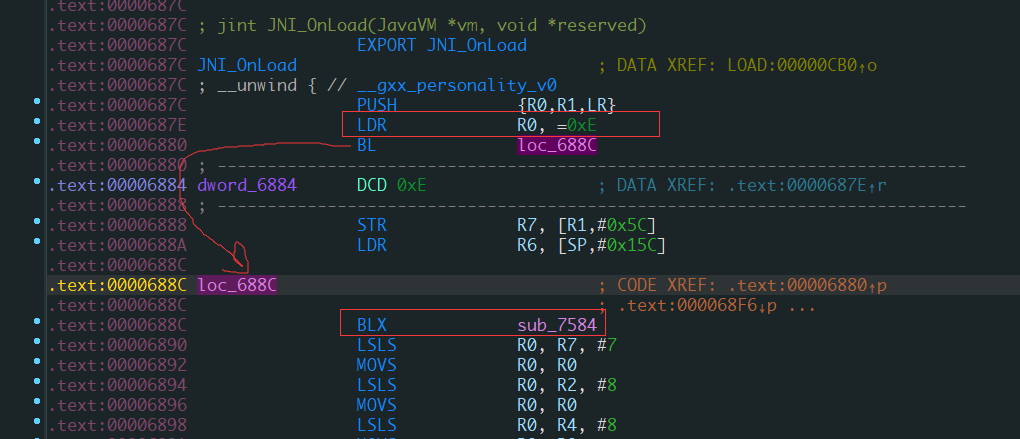
最大的特征就是赋值r0然后跳转到分发器（通常用LR和R0配合计算跳转表地址）。每个块结构类似: 代码段，给r0赋值，跳转到分发器 -> 下一个块。

对抗这种混淆，与CFF类似，遍历跳转表，直接patch跳转地址到真实块。

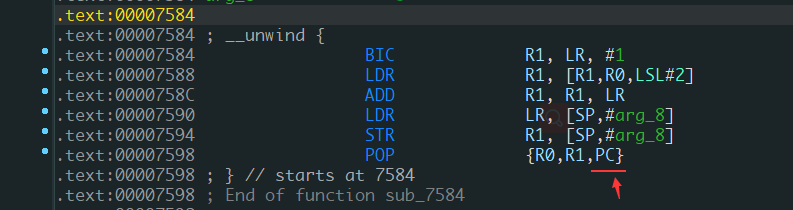
# 快手

首先在导出表找到JNI\_OnLoad函数。

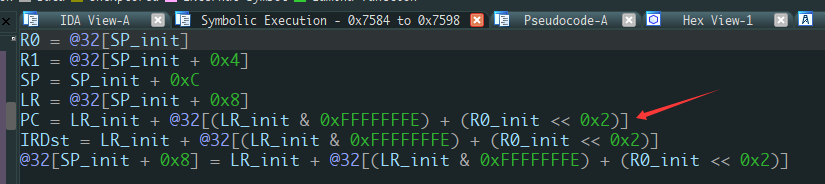
可以看到满足的一些基本特征R0赋值，BL跳转



去对应的sub\_7584确定一下，这里最明显的特征是POP了一下PC



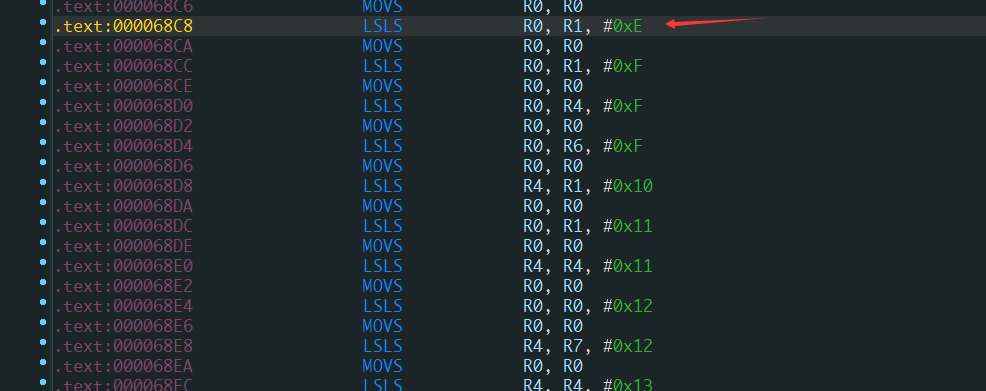
符号执行看下这段代码的执行结果



把LR=0x6890和R0=0xE代入进去，计算一下

PC = LR\_init + @32[(LR\_init & 0xFFFFFFFE) + (R0\_init << 0x2)] = 0x6890+@32[0x68c8]

看下0x68c8对应的值，这里结构和前面BLX sub\_7584后面的类似。很明显这里应该是IDA解析错了，把内嵌的data段当成了code段

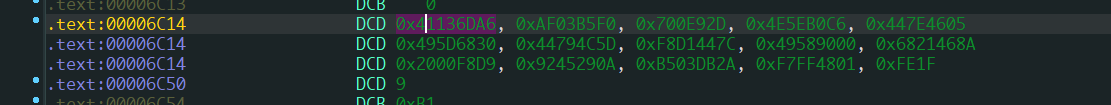


在0x686c按D解析成数据，得到0x388

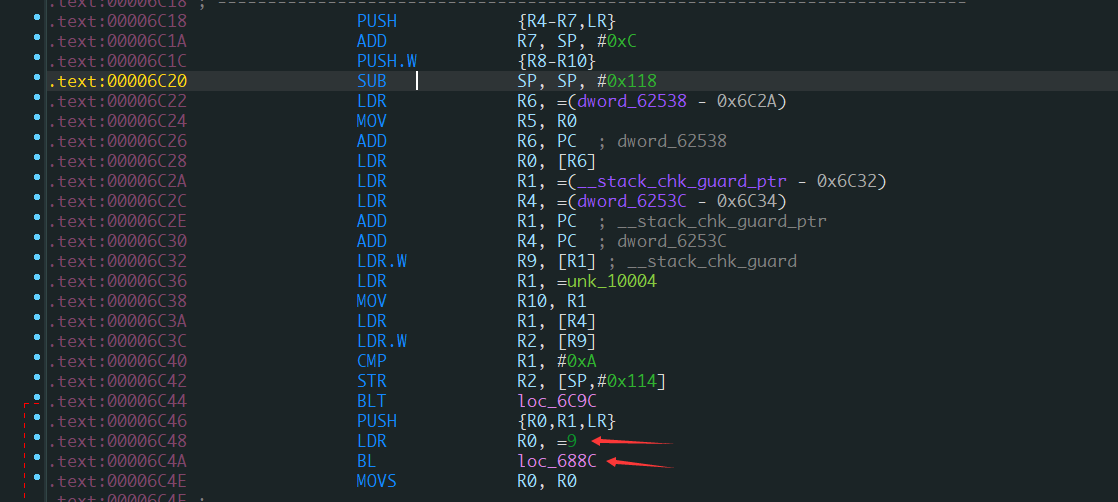


计算到下一个块的跳转地址为0x6890+0x388 = 0x6c18

可以看到，由于IDA使用递归反汇编，这里的code端未能被反编译，以达到对抗递归反编译的静态分析



按C转化为汇编之后，可以看到在末尾，又是R0赋值，BL跳转到loc\_688c,



到这里，我们基本可以判断loc\_688c是一个分发器，每个片段末尾都会通过这个分发器跳转到下一个代码片段。

接下来，我们编写一个idaPython的脚本来patch相应汇编进行修复。