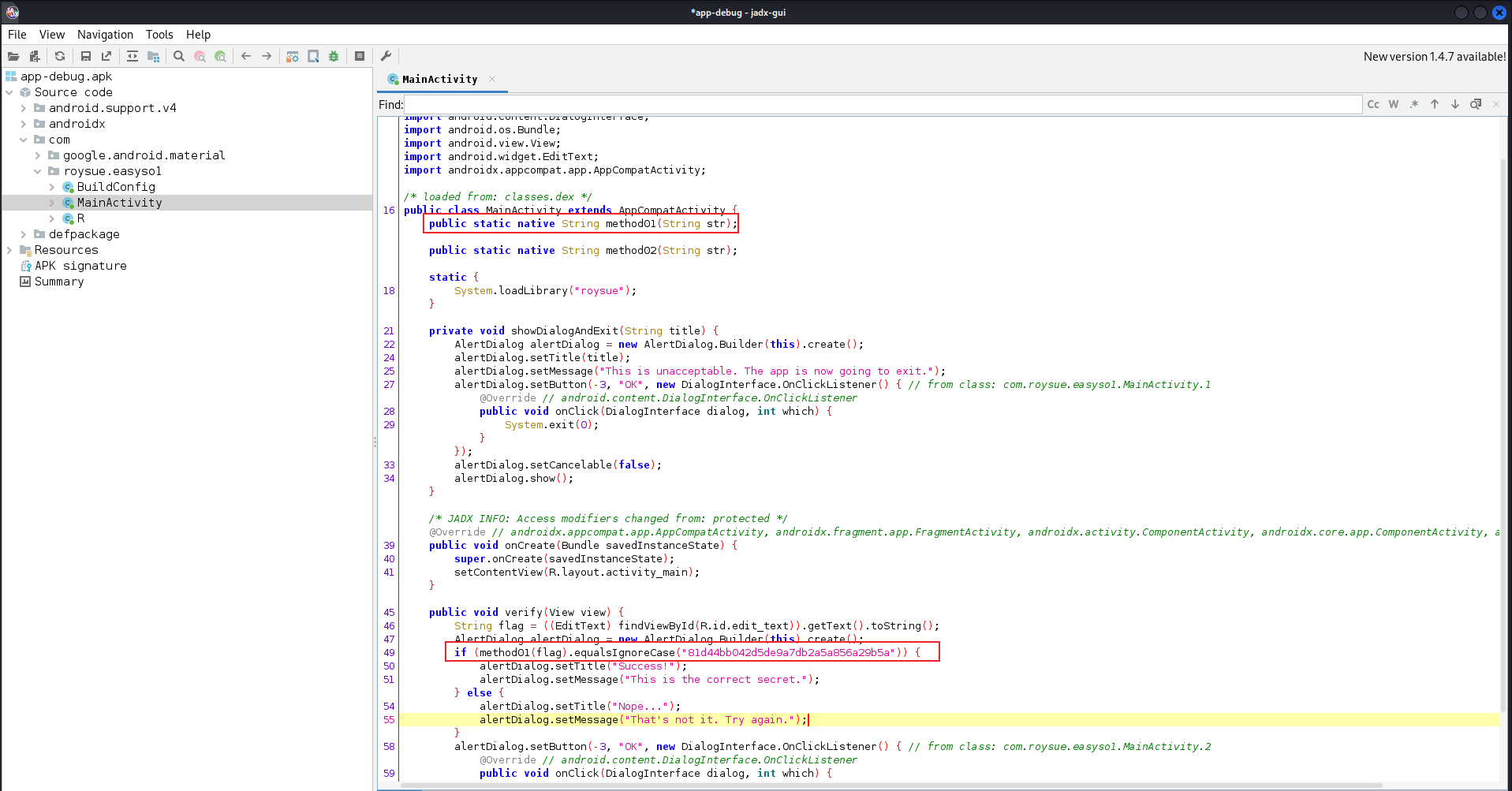
# easyso解题思路

### 0x01 前言

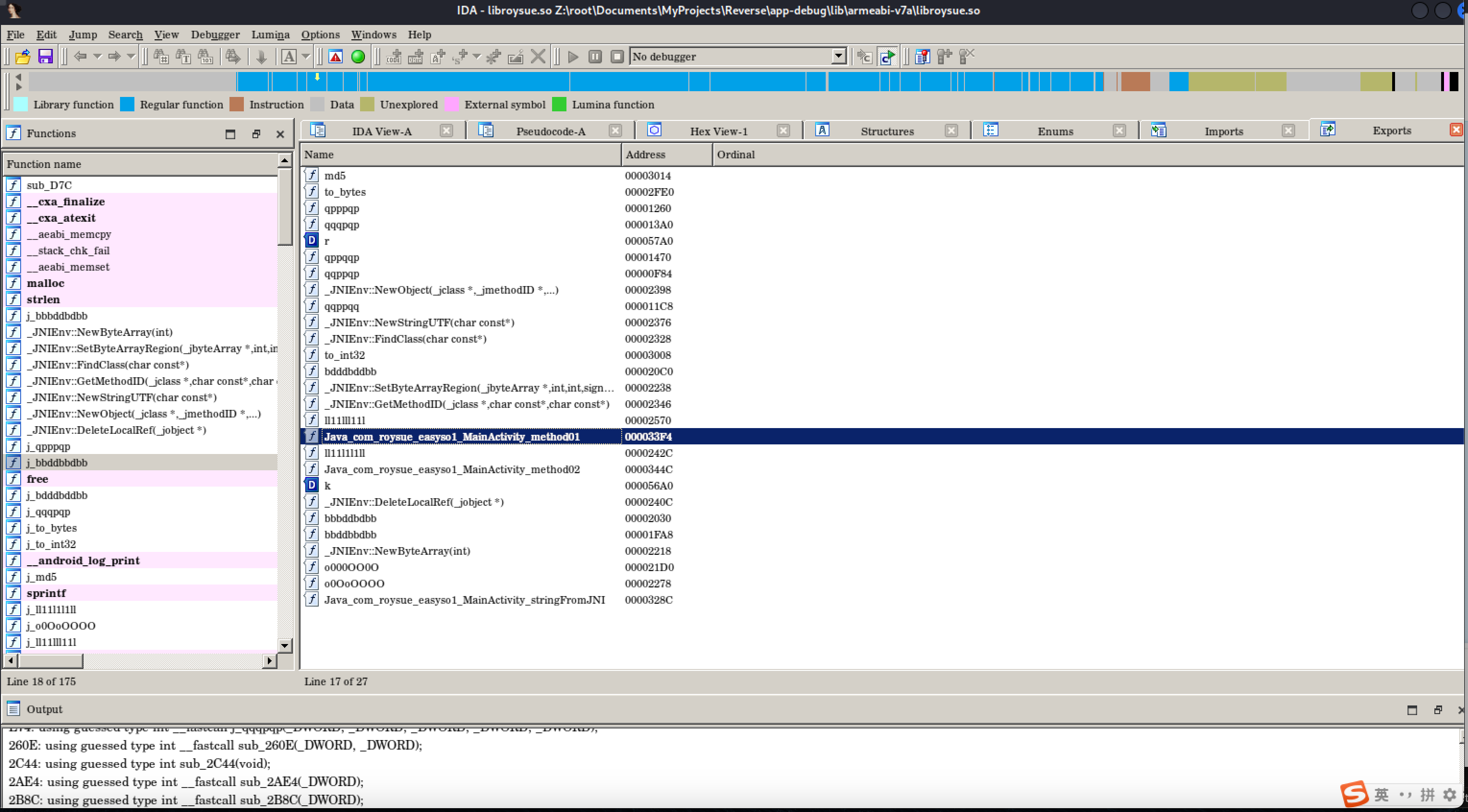
作为一个菜鸡，很久没有搞so分析了，碰巧看到r0ysue大佬发的朋友圈，看到easy这个字样，寻思着可以康康，整体而言，没有加固，so中混淆手段较为温和（基本没有），用frida可以解题很友好（看源码死磕很恼火）。

### 0x02 分析思路

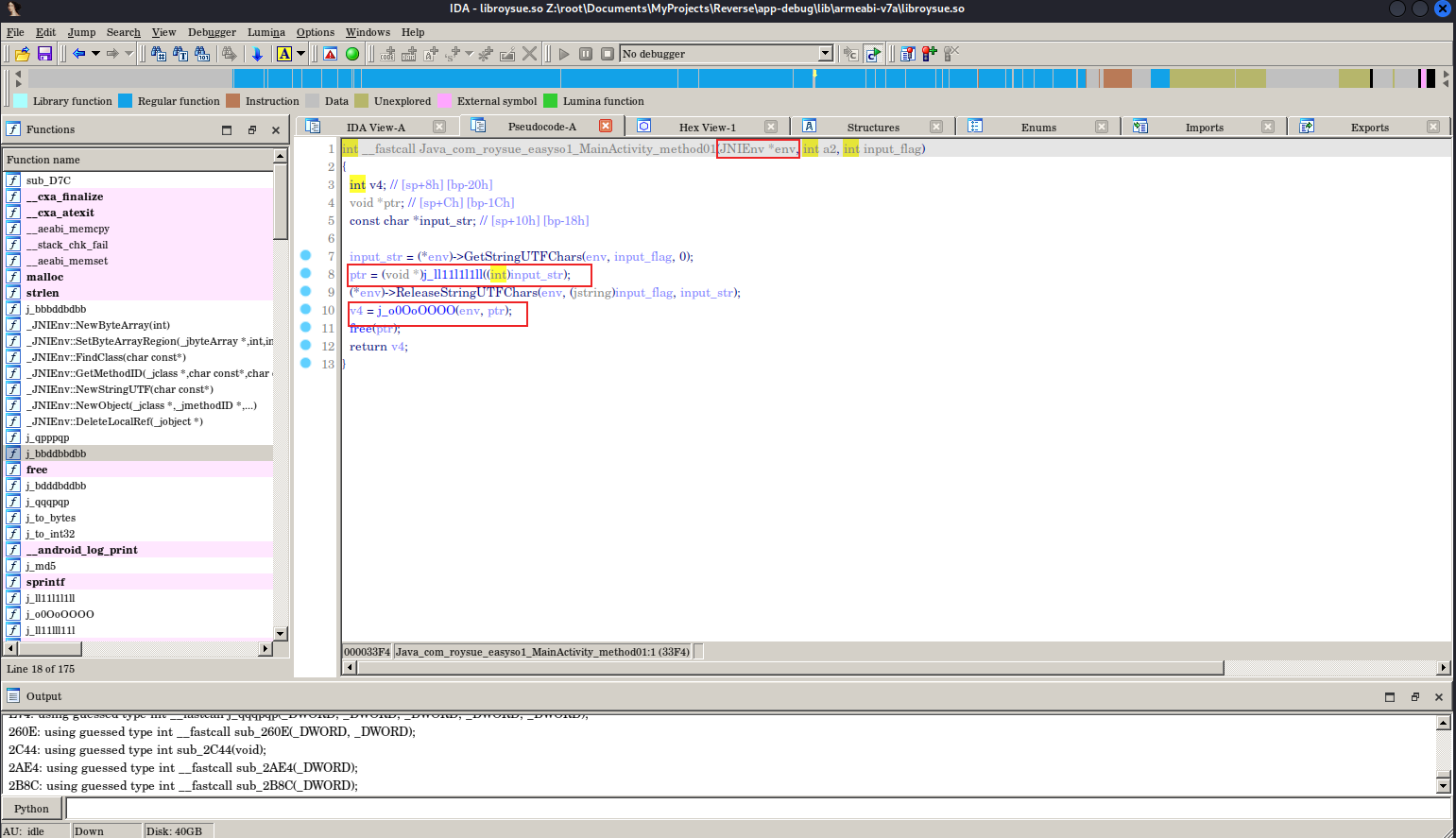
先把app扔到jadx中查看代码，发现程序将输入的字符串函数调用了so中的method01函数计算后与字符串81d44bb042d5de9a7db2a5a856a29b5a 作对比，隐私核心逻辑在so文件中的method01函数中。



ida打开libroysue.so文件，查看导出表Exports，找到method01对应的函数为Java\_com\_roysue\_easyso1\_MainActivity\_method01



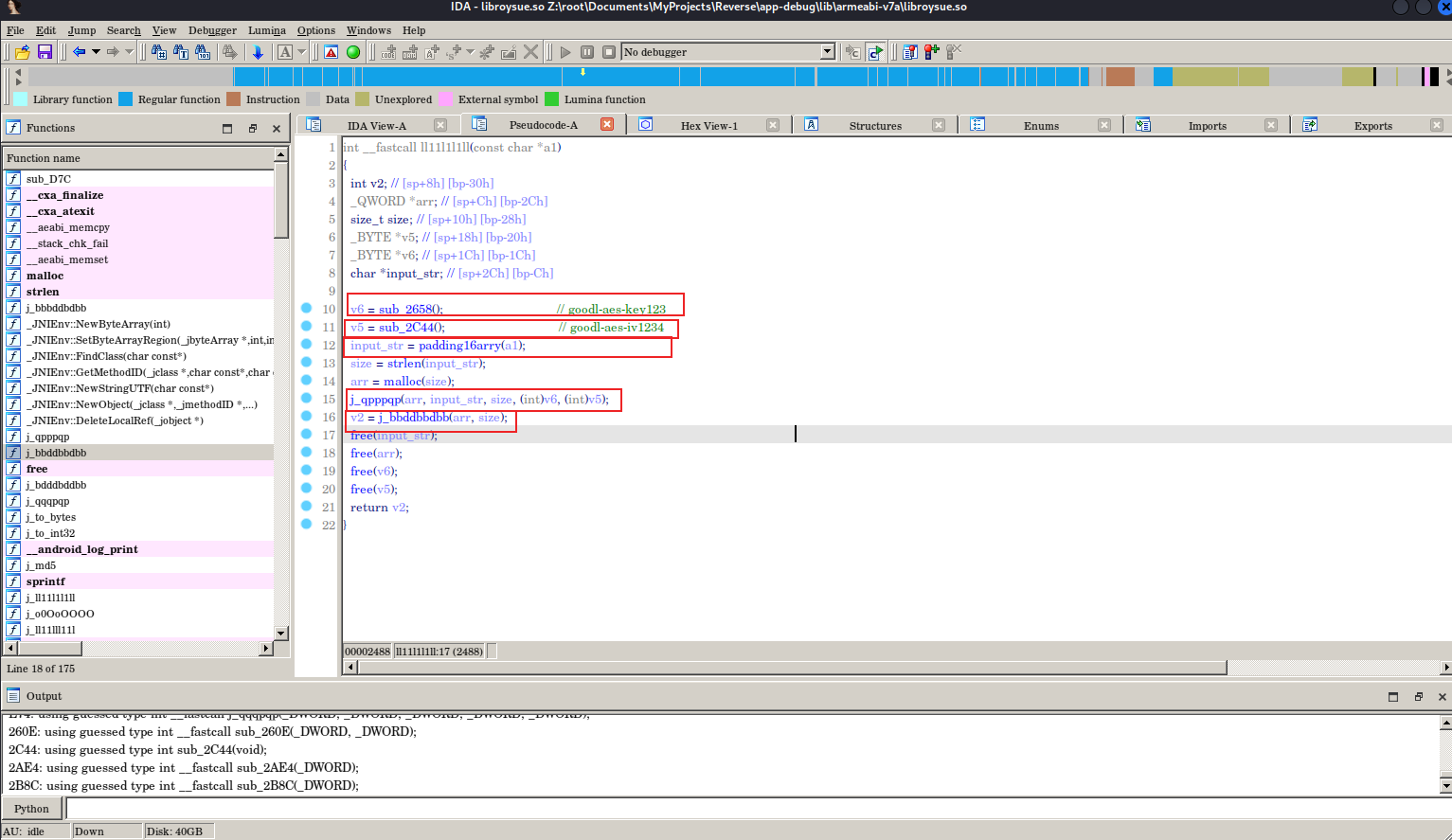
F5反编译为C语言，修改a1的类型为JNIEnv\* 可使代码更加直观。



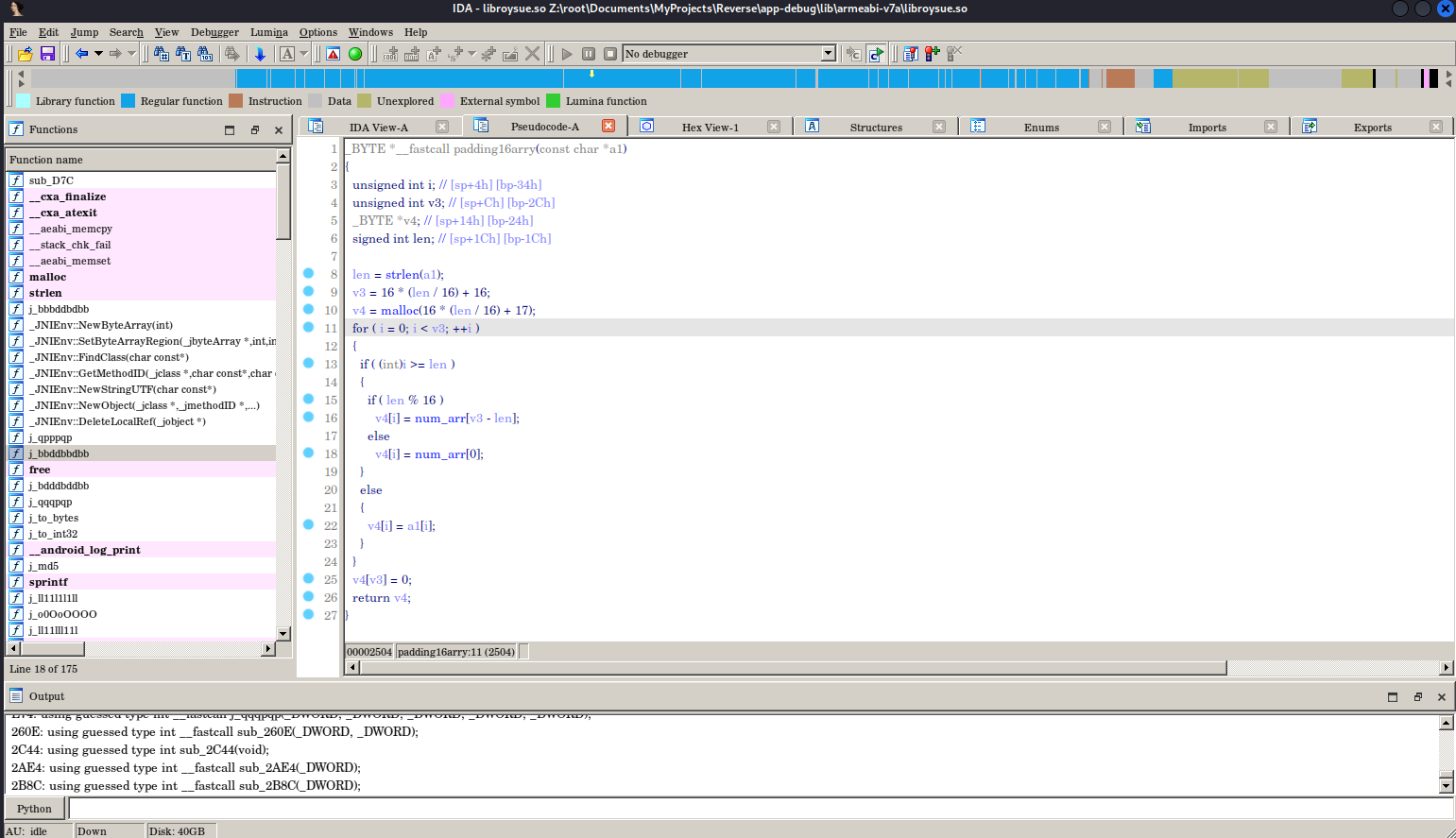
分析method01函数代码可知，函数对传入字符先进行格式转化，然后扔到ll11l1l1ll这个函数中处理

于是继续分析ll11l1l1ll这个函数，发现处理逻辑如下：

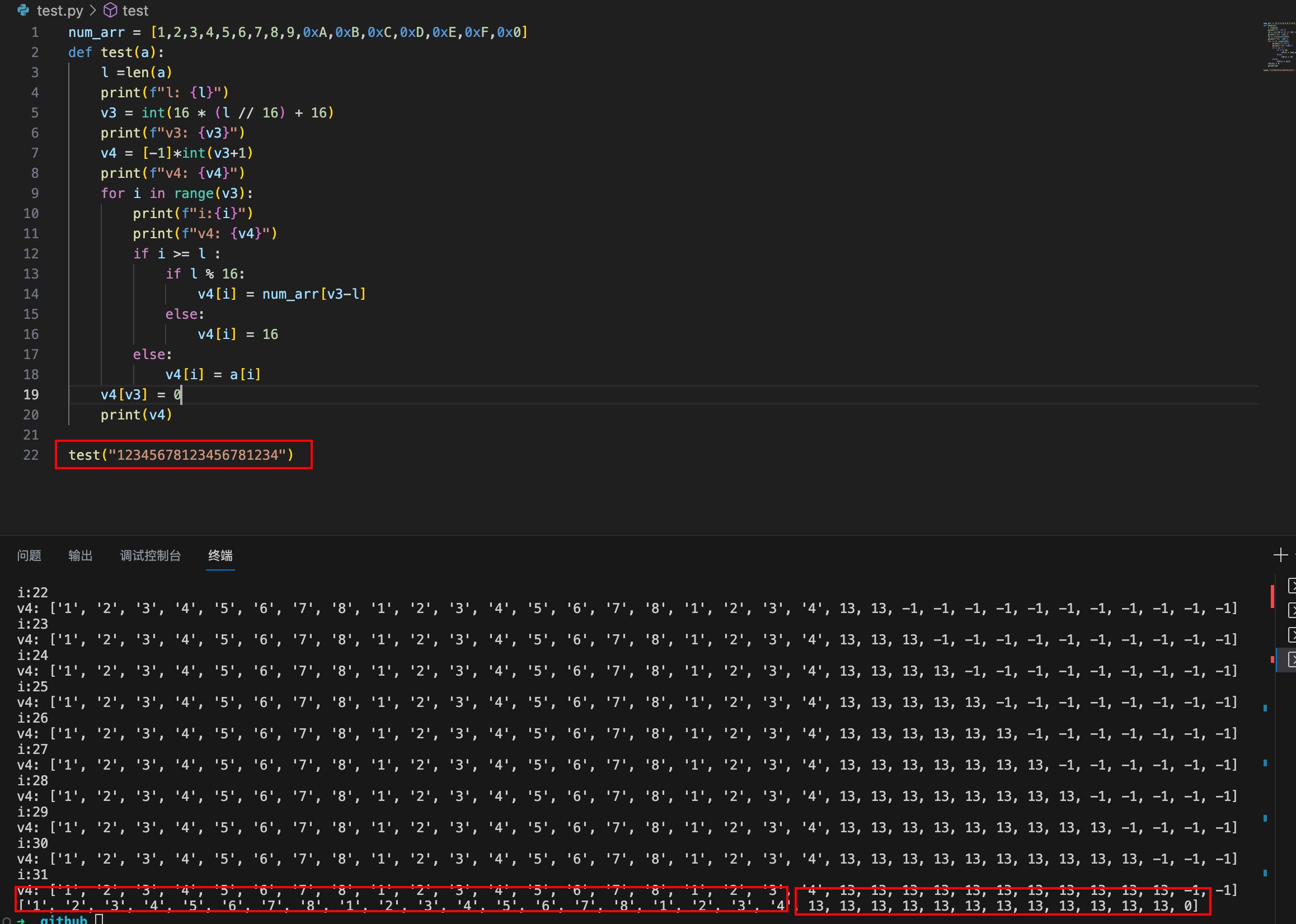
* 调用sub\_2658：函数无输入，猜测返回固定值
* 调用sub\_2C44：函数无输入，猜测返回固定值
* padding16array函数对a1进行处理（函数名称为人为更改）
* 调用qpppqp函数
* 调用bbddbbdbb函数



sub\_2658，sub\_2C44两个函数既然是固定返回值，则没必要分析代码，等会儿直接用frid进行hook返回值即可，所以先分析padding16array函数，该代码不长，分析完大概就是将数据进行长度补齐，补齐为16对整数倍。



用python复写代码实验了一下，结果如下，当输入为20位时，会补齐至32位，填充为32-20=12，及为arr中第12个数字，即为0xD（十进制的13），因此重命名函数为padding16array：



接下来写hook脚本，查看sub\_2658，sub\_2C44这两个函数的返回值，hook代码如下：

function hook\_sub2658() {

var so\_addr = Module.findBaseAddress("libroysue.so");

var sub2658\_addr = so\_addr.add(0x2658 + 1)

Interceptor.attach(sub2658\_addr, {

onEnter: function(args) {},

onLeave: function(retval) {

console.log("sub2658 ret: ", retval.readCString())

},

})

}

function hook\_sub2C44() {

var so\_addr = Module.findBaseAddress("libroysue.so");

var sub2C44\_addr = so\_addr.add(0x2C44 + 1)

Interceptor.attach(sub2C44\_addr, {

onEnter: function(args) {},

onLeave: function(retval) {

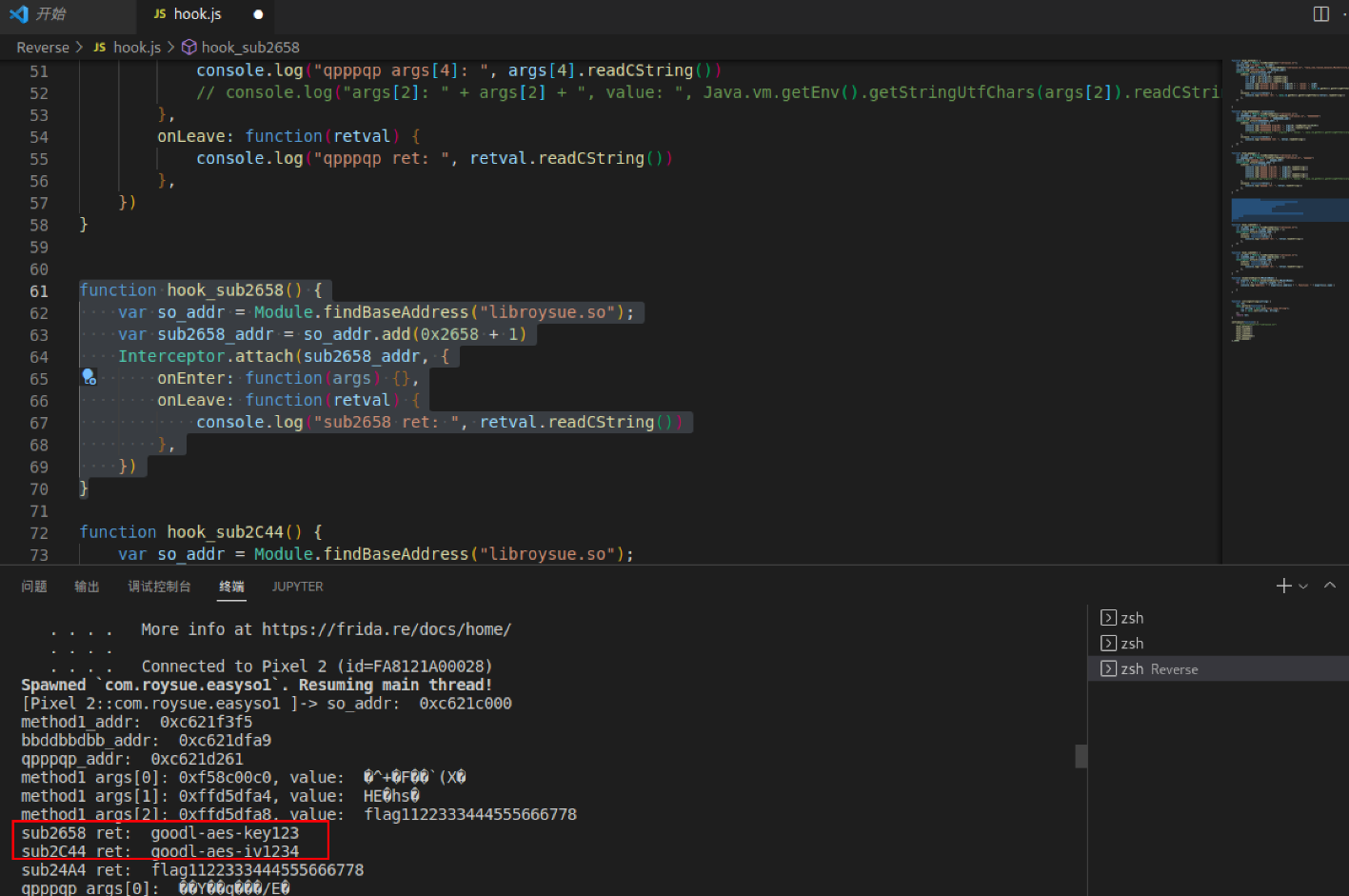
console.log("sub2C44 ret: ", retval.readCString())

},

})

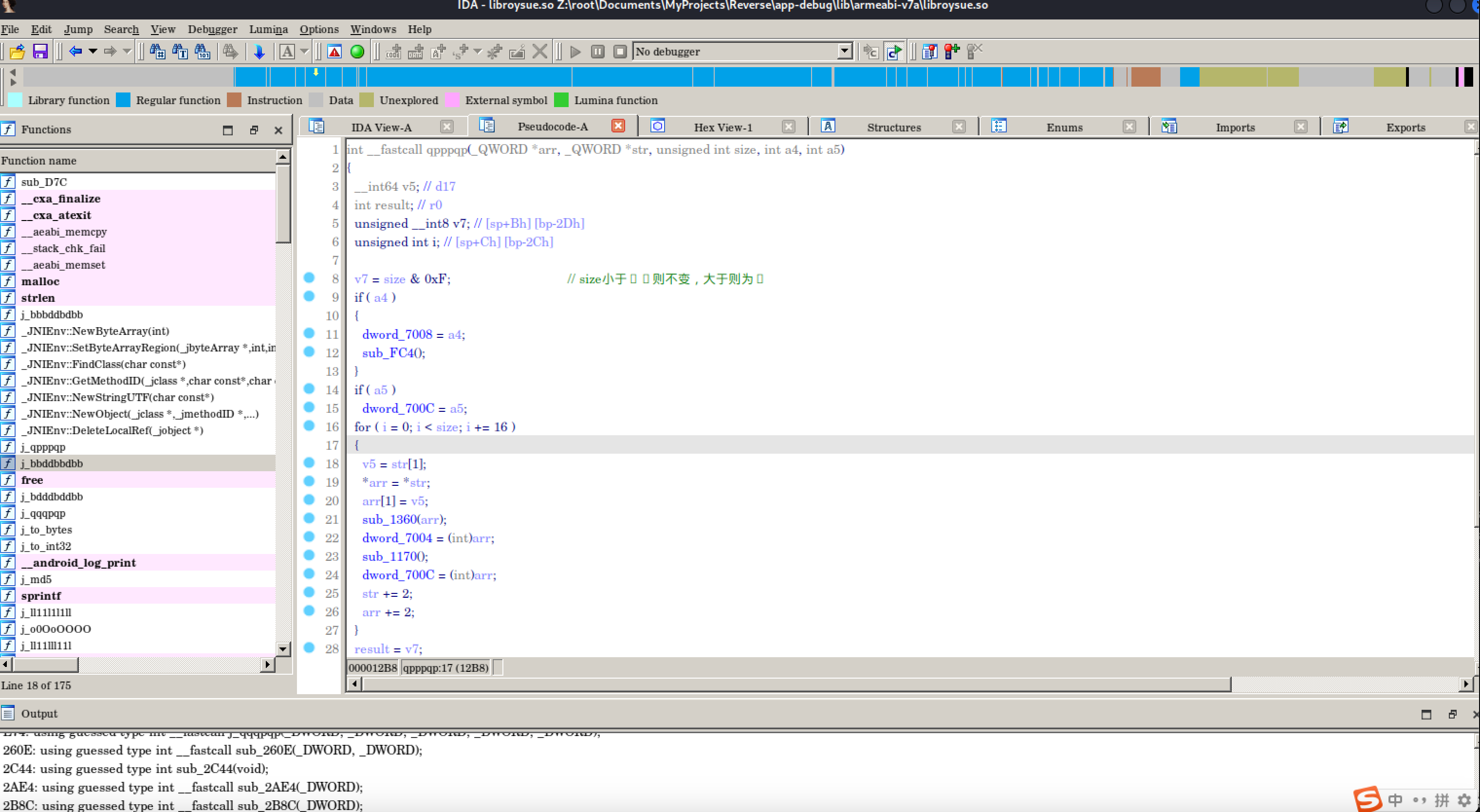
}

hook结果如下，sub2658返回值为goodl-aes-key123，sub2C44为goodl-aes-iv1234：



根据两个函数的返回值来看，可以猜测下面应该是进行了aes加密，因为涉及到key和iv（即密钥和偏移量），应该进行的是cbc模式加密。

那么根据函数的输入值，可以猜测，qpppqp应该是进行aes加密的函数



查看qpppqp 对代码，发现代码里面还有其他函数嵌套，寻思着都用上frida了，可以直接hook一下输入输出，验证一下猜想，若不正确再细细分析，hook代码如下：

function hook\_qpppqp() {

var so\_addr = Module.findBaseAddress("libroysue.so");

var qpppqp\_addr = Module.findExportByName("libroysue.so", "qpppqp")

console.log("qpppqp\_addr: ", qpppqp\_addr)

Interceptor.attach(qpppqp\_addr, {

onEnter: function(args) {

console.log("qpppqp args[0]: ", args[0].readCString())

console.log("qpppqp args[1]: ", args[1].readCString())

console.log("qpppqp args[2]: ", args[2])

console.log("qpppqp args[3]: ", args[3].readCString())

console.log("qpppqp args[4]: ", args[4].readCString())

// console.log("args[2]: " + args[2] + ", value: ", Java.vm.getEnv().getStringUtfChars(args[2]).readCString())

},

onLeave: function(retval) {

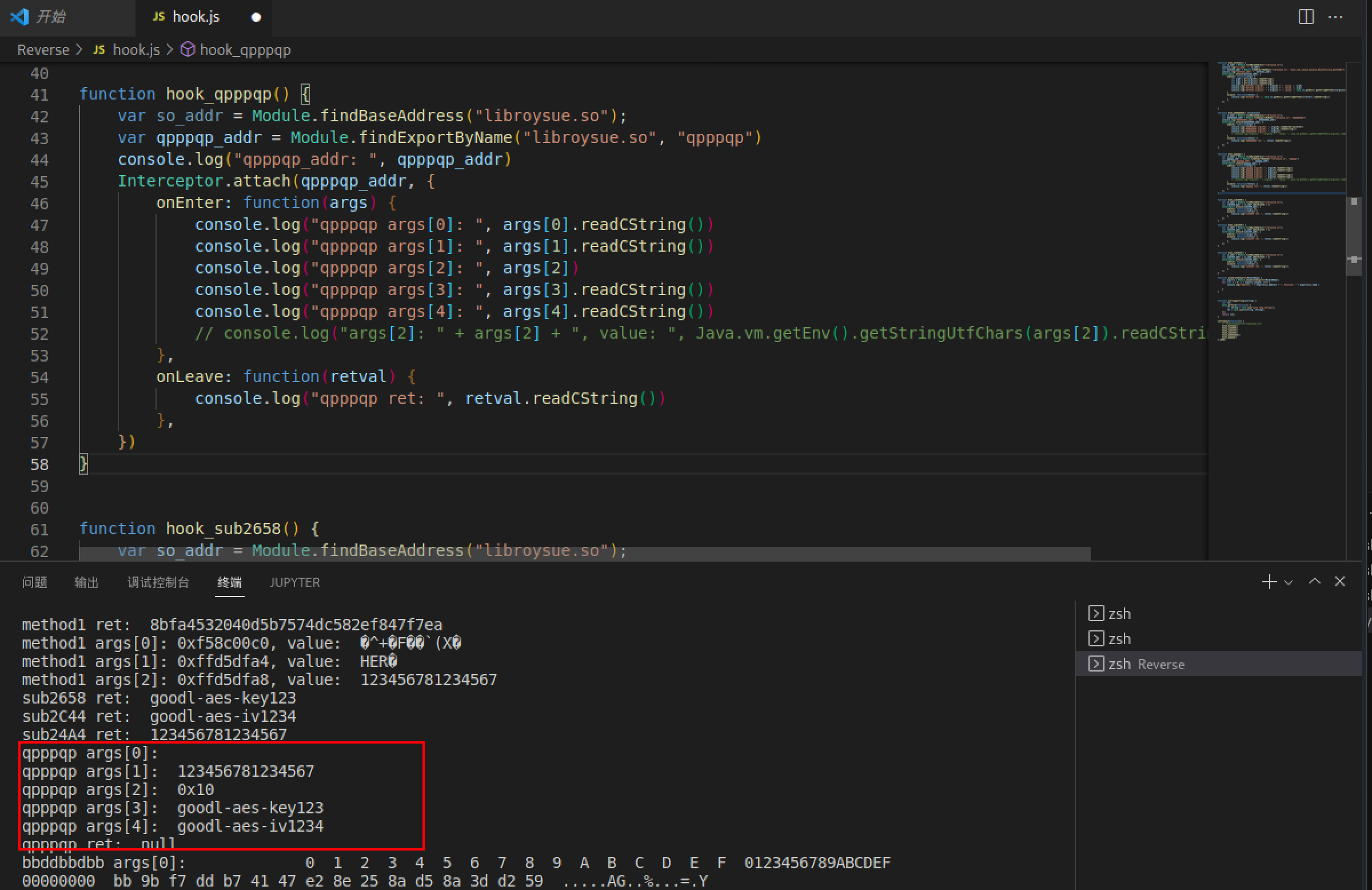
console.log("qpppqp ret: ", retval.readCString())

},

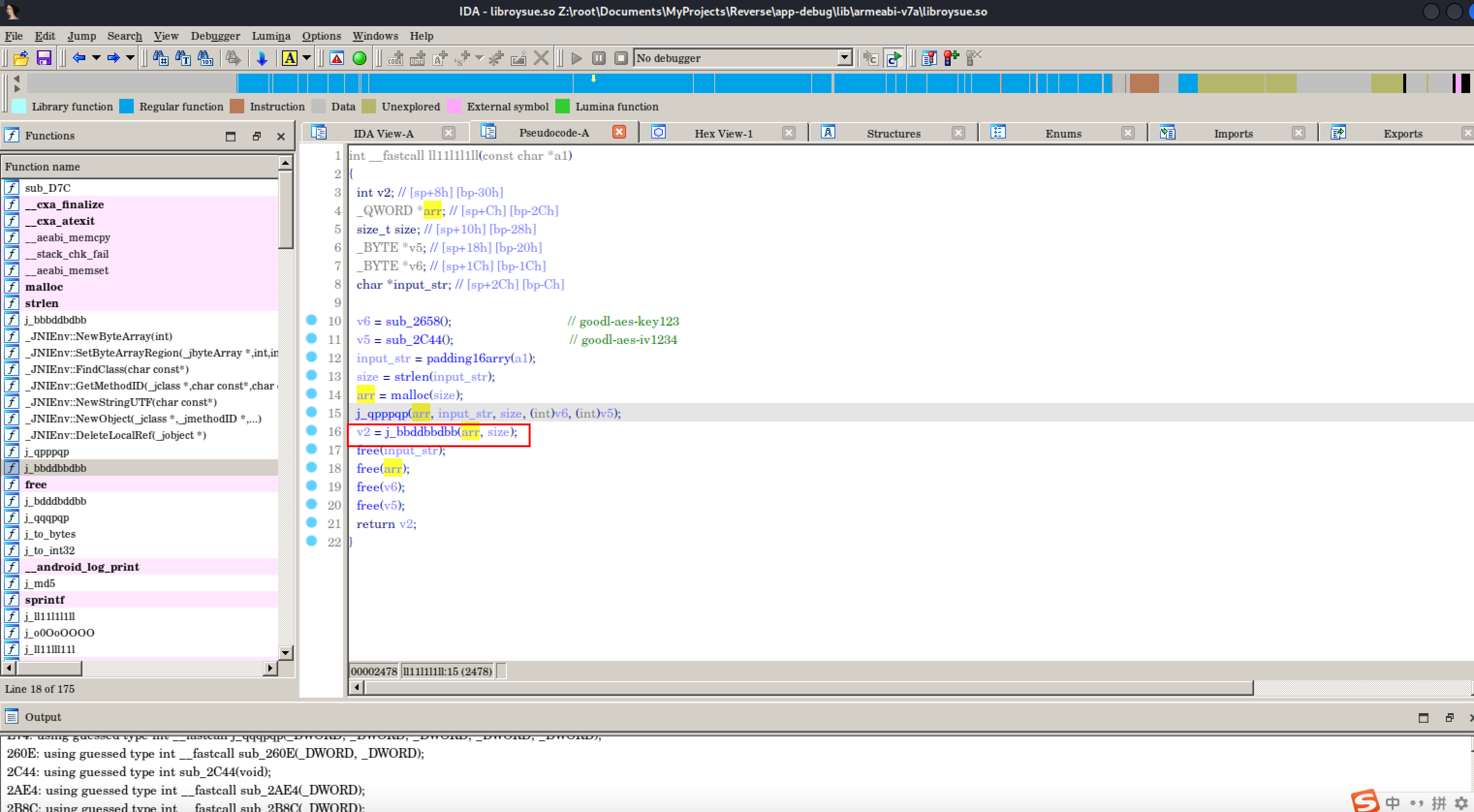
})

}

hook结果如下：



发现返回结果是null，有点懵逼，核对了一下函数应该是有返回值的才对。不过不影响，看了下输出结果应该再arr里面，而arr又是bbddbbdbb函数的输入，因此可以查看该函数的输入值来确定



bbddbbdbb 函数hook代码如下：

function hook\_bbddbbdbb() {

var so\_addr = Module.findBaseAddress("libroysue.so");

var bbddbbdbb\_addr = Module.findExportByName("libroysue.so", "bbddbbdbb")

console.log("bbddbbdbb\_addr: ", bbddbbdbb\_addr)

Interceptor.attach(bbddbbdbb\_addr, {

onEnter: function(args) {

console.log("bbddbbdbb args[0]: ", args[0].readByteArray(0x30))

console.log("bbddbbdbb args[0]: ", args[0].readCString())

console.log("bbddbbdbb args[1]: ", args[1])

// console.log("args[2]: " + args[2] + ", value: ", Java.vm.getEnv().getStringUtfChars(args[2]).readCString())

},

onLeave: function(retval) {

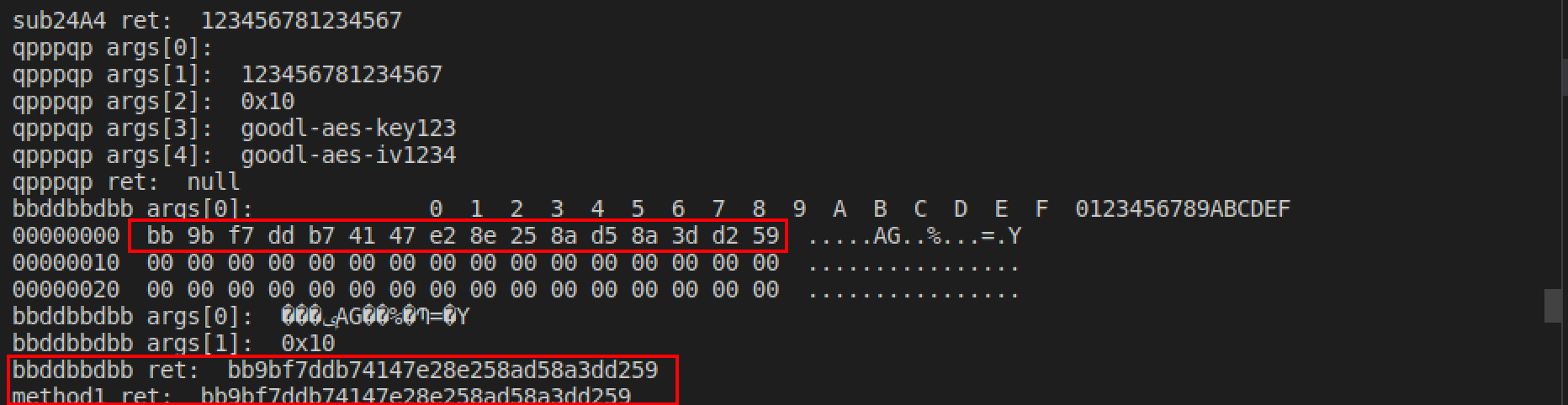
console.log("bbddbbdbb ret: ", retval.readCString())

},

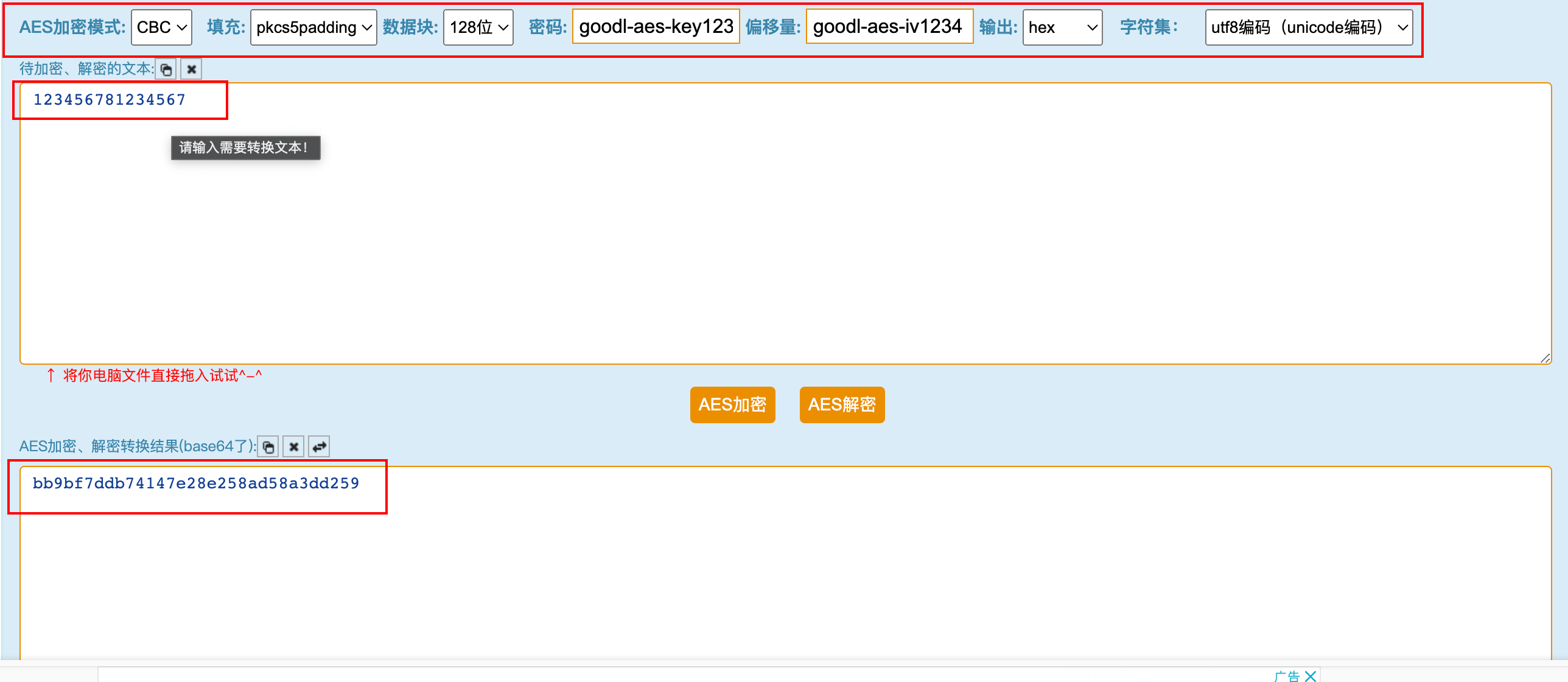
})

}

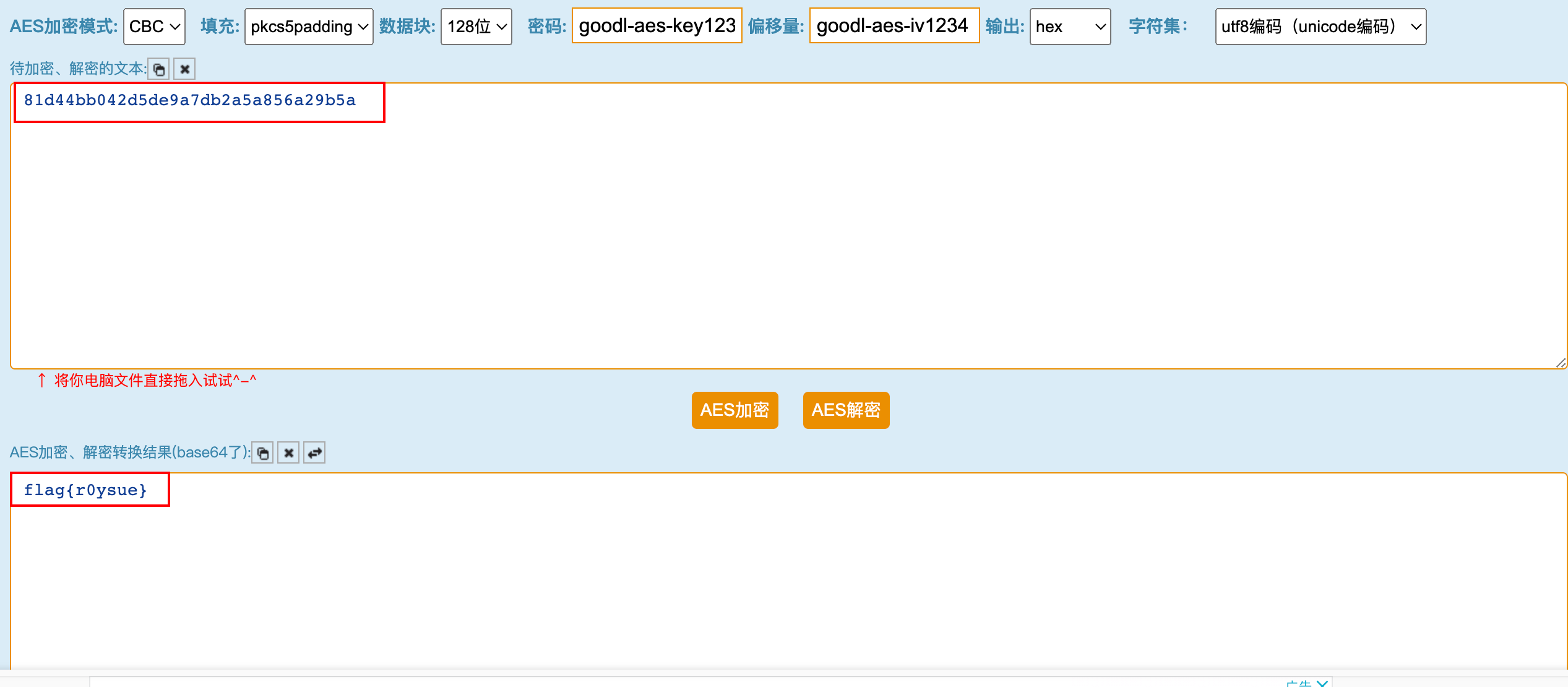
在输入值为123456781234567下，输出值为bb9bf7ddb74147e28e258ad58a3dd259：



也就是说函数qpppqp 对输出就是最终method1的输出，拿这个结果和标准aes加密结果对比一下



发现结果相同，于是将对比用的字符串用相同参数进行解密，可得到flag:flag{r0ysue}



### 0x03 后记

在尝试函数是否为aes加密函数的时候做了几次尝试，包括明文的长度为什么是15，因为分析qpppqp函数是，第一行有一个size & 0xF ，进行了一下实验，发现最多取到15，而且大于15的明文加密的长度明显长很多，所以最终用15长度的明文进行实验。