[原创]某企业级加固[四代壳]VMP解释执行+指令还原 精

贝a塔 3

大牛

2020-1-5 05:42

现在的VMP的比较常见了，应该也是稳定性满足要求了，今天来分析一波，如有不当还请各位大佬指正
实际上 libdexjni.so在不同的APP中体积会不一样，应该是硬编码写入字符串和指令导致的

1-VMP还是先看下opcode部分知识，DEX指令格式

代码转换成DEX指令先看代码

```
public class BaseActivity extends FragmentActivity {


    public Resources getResources() {
        return super.getResources();
    }
```

对应的第一条指令是

```
6F 10 1C 33 01 00 0C 00 11 00 0(
```

每条指令是2字节，所以先看第一条 6f 20,根据官方文档 6F的解释是 invoke-super 格式为35c

A|G|*op* BBBB F|E|D|C	**35**c	*[A=5]* **op** {vC, vD, vE, vF, vG}, meth@BBBB *[A=5]* **op** {vC, vD, vE, vF, vG}, site@BBBB *[A=5]* **op** {vC, vD, vE, vF, vG}, type@BBBB *[A=4]* **op** {vC, vD, vE, vF}, **kind**@BBBB *[A=3]* **op** {vC, vD, vE}, **kind**@BBBB *[A=2]* **op** {vC, vD}, **kind**@BBBB *[A=1]* **op** {vC}, **kind**@BBBB *[A=0]* **op** {}, **kind**@BBBB

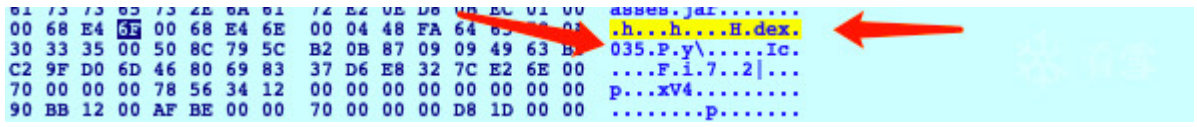
A | G | op BBBB F|E|D|C

根据opcode克制总共6个字节，对应的就是
A=1 G=0 op=6f BBBB就是 331c,然后是C=1 D=0 E=0 F=1
  所以这里转换过来就是

```
invoke-super {p0}, Landroidx/fragment/app/FragmentActivity;->getResources()Landroid/content/res/Resources;
```

2-反调试

通过常规手段，在关键的open函数观察，然后逆向查找
发现几处反调试
0x47CAC 处是创建线程，检测运行时间，getpid 然后 linux_eabi_syscall(__NR_kill, a1, a2)来杀死进程
0x047C70 处是cmdline反调试，https://bbs.pediy.com/thread-223460.htm 这位大佬提到过
0x489EC 处是 /proc/status检测反调试
实际可能还有，但是在找到这三处之后，我发现特殊的地方是刚好在JNI_OnLoad处有个总的入口，所以直接nop指令反调试就gg了
我用 arm64调试的 mov w1,w1 对应的的hex是E103012A
然后dump出dex,先内存找到dex.035



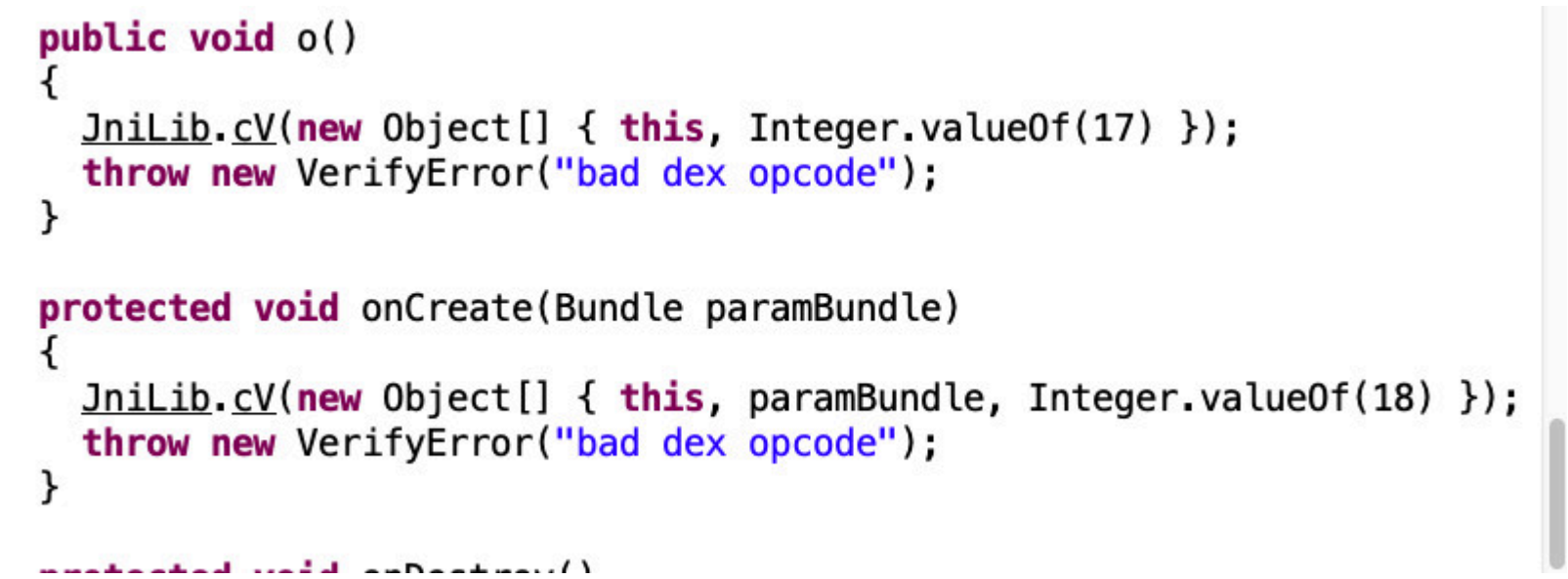

```python
import struct


start = 0x75172191ec

dump_so = "/Users/beita/tmp/bangbang/dump_vmp.dex"
length = 0x6ee27c
file = open(dump_so,'w')
file.close()
fn = AskStr(dump_so ,"save as:")
with open(fn,"wb+") as f:
    for addr in range(start , start+length):
        f.write(struct.pack("B" , Byte(addr)))
    print "success to save as "
```

3-VMP的具体分析

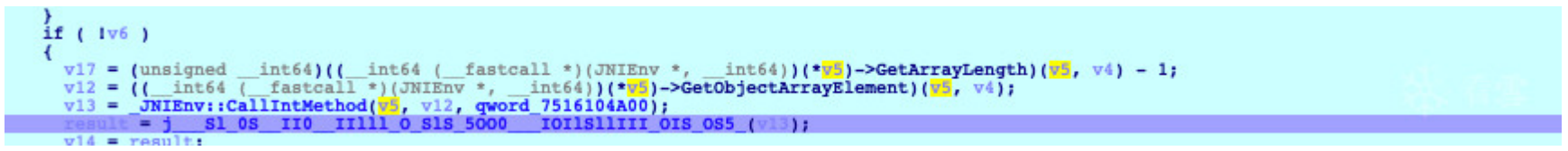
得到dex之后，转成jar，看了下，大部分函数是 JniLib.cV等来做的，但是有一个Integer.valueof，是一个函数索引，用来查找指令的



```
public void o()
{
  JniLib.cV(new Object[] { this, Integer.valueOf(17) });
  throw new VerifyError("bad dex opcode");
}

protected void onCreate(Bundle paramBundle)
{
  JniLib.cV(new Object[] { this, paramBundle, Integer.valueOf(18) });
  throw new VerifyError("bad dex opcode");
}
```

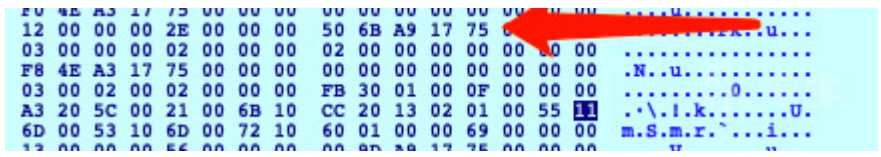
附加调试发现实际在这里解开这个java数组也就是 new Object的这个数组



这里用onCreate来分析 索引是18=0x12

JniLib.cV(new Object[] { this, paramBundle, Integer.valueOf(18) });

调试往下走，根据这个索引，会取出一个结构体信息,结合上下文信息



这里取出 0x7517a96b50的值 是 0x12


```
strut JavaInfo {
    uint32_t index;     // 0x12   这是java层传递的
    uint32_t unknow2;   // 0x2e   未知
    uint64_t dexcode;   // dexcode指针
    uint32_t unknow4;   // 0x03
    uint32_t unknow5;   // 0x02
    uint32_t unknow6;   // 0x02  这里看起来没有用到   但是貌似是DexCode的内容
};
```

跳转到dexcode的位置看下内容

```
000075172E6710  03 00 00 00 02 00 00 00  02 00 00 00 00 00 00 00  ................
000075172E6720  60 BF 28 17 75 00 00 00  00 00 00 00 00 00 00 00  `.(.u...........
000075172E6730  03 00 02 00 02 00 00 00  FB 30 01 00 0F 00 00 00  ........0......
000075172E6740  A3 20 5C 00 21 00 6B 10  CC 20 13 02 01 00 55 11  . \.!.k.....U.
000075172E6750  6D 00 53 10 6D 00 72 10  60 01 00 00 69 00 00 00  m.S.m.r.`...i...
000075172E6760  13 00 00 00 56 00 00 00  00 3A 2E 17 75 00 00 00  ....V....:..u...
000075172E6770  03 00 00 00 01 00 00 00  02 00 00 00 00 00 00 00  ................
000075172E6780  68 BF 28 17 75 00 00 00  00 00 00 00 00 00 00 00  h.(.u...........
000075172E6790  14 00 00 00 18 00 00 00  80 60 33 17 75 00 00 00  .........`3.u...
```

加密的DexCode的内容

```
struct DexCode {
    u2  registersSize;   // 3
    u2  insSize;           、、 2
    u2  outsSize;
    u2  triesSize;
    u4  debugInfoOff;       /* file offset to debug info stream */
    u4  insnsSize;          /* size of the insns array, in u2 units */
    u2  insns[1];
    }
```

registerSize = 3

insSize = 2

outsSize = 0

.....

主要看

insnsSize = 0xf

共15条指令 ，但是这个指令不是 标准的dex指令 opcode被改过，且字符串信息也是被改过，就是是说他不是系统来解析的，而且会有一个对应关系

A3 20 5C 00 21 00 6B 10 CC 20 13 02 01 00 55 11

6D 00 53 10 6D 00 72 10 60 01 00 00 69 00

进入到vm_parse函数之前的代码还能F5看下逻辑，但是到 vm_parse地址是29b70位置处，F5不好用了，貌似是刻意把这个函数写的非常大，

有点像dalvik里边的HANDLE那种搞到一起， 这样在加固过程中OLLVM混淆之后，更加复杂

在解析opcode之前会进行数据保存

信息看起来是保存到一组结构中

```
struct Infos1{
    uint64_t data1;
    uint64_t *data2;  // data2 = malloc(32)    是根据JavaInfo的dexCode来的
    uint64_t data3;
    uint64_t data4;
    uint64_t data5;
    uint64_t data6;
    uint64_t data7;
    uint64_t data8;    // JavaInfo的data3的值
};
```

调试继续往下走，来到 j___SI_I5_IO000_0SSIO_I0_O__OI_5I___ISSI0_IO5_0I5I5S5_ 这个函数，这个函数不能F5了，要根据汇编来分析具体的vm是如何

解析opcde来实现代码运行的

最终的 入口是 29b70这个函数

调用获取GetMethodID的过程是

vm_parse 29b70 - 29bb0 - 4ae80 - 4aeb4 - 4e78c - 3f92c 开始获取名称和GetMethodID

第一个参数 结合全局变量可以获得这些内容Class MethodSig MethodName

前面提到，vmp可能会借助jni来实现，所以现在GetMethodID下段点，查看数据，方法名称和签名

```
.text:0000007517CF9980 LDR             X8, [X21]
.text:0000007517CF9984 LDR             X1, [X19]
.text:0000007517CF9988 LDR             X8, [X8,#0x30]          ; FindClass  参数是  android/xxx/FraqmentActivity
.text:0000007517CF998C MOV             X0, X21
.text:0000007517CF9990 BLR             X8
.text:0000007517CF9994 LDR             X8, [X21]
.text:0000007517CF9998 LDP             X3, X2, [X19,#8]
.text:0000007517CF999C LDR             X8, [X8,#0x108]         ; GetMethodID onCreate
.text:0000007517CF99A0 MOV             X9, X0
.text:0000007517CF99A4 STR             X9, [SP,#0x15A0+var_7D8]
.text:0000007517CF99A8 MOV             X0, X21
```

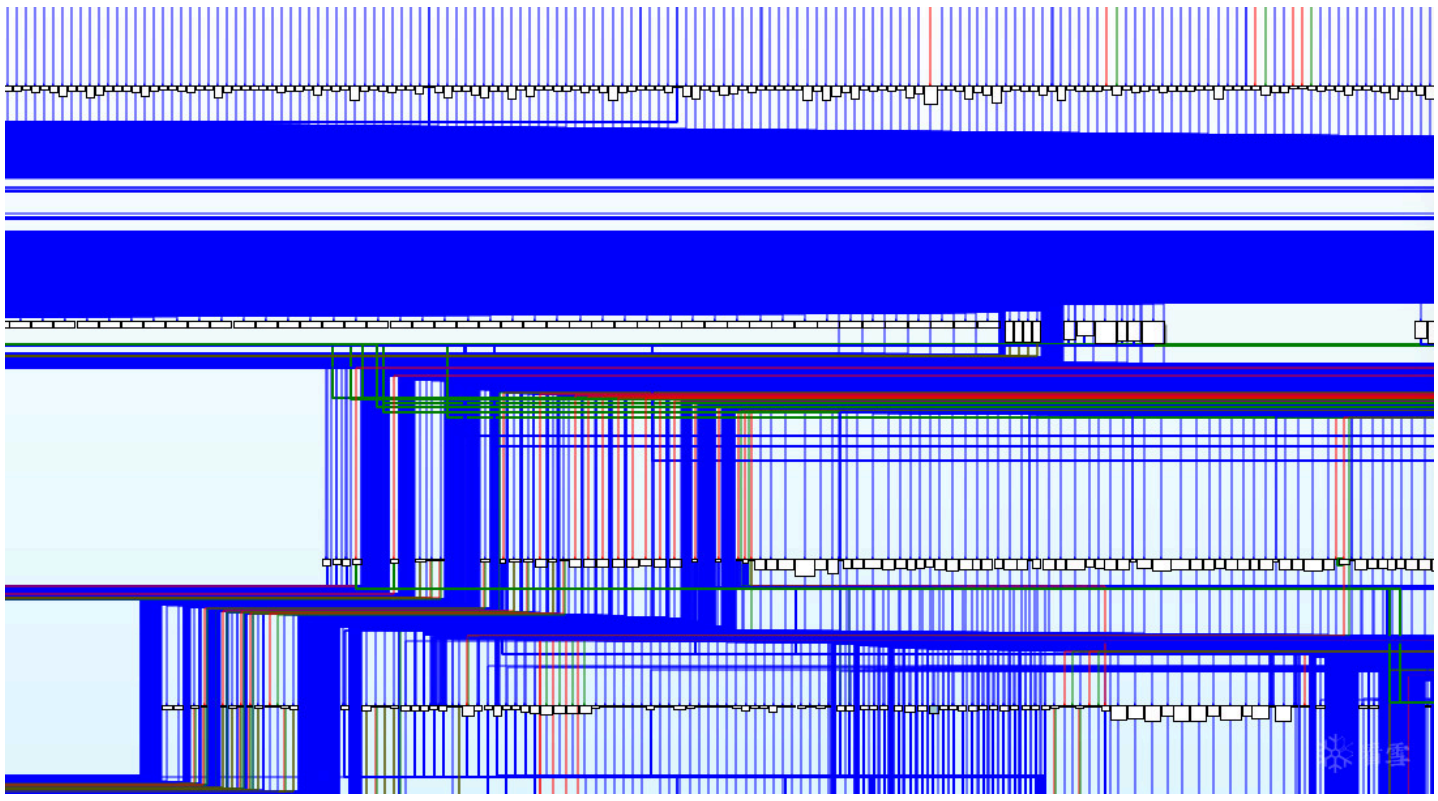
寄存器x2

```
0000007517B09A50  6F 6E 43 72 65 61 74 65  00 00 00 00 00 00 00 00  onCreate........
0000007517B09A60  50 9A B0 17 75 00 00 00  08 00 00 00 00 00 00 00  P...u...........
```

寄存器x3

```
0000007515E0EC60  28 4C 61 6E 64 72 6F 69  64 2F 6F 73 2F 42 75 6E  (Landroid/os/Bun
0000007515E0EC70  64 6C 65 3B 29 56 00 00  00 00 00 00 00 00 00 00  dle;)V..........
```

但是因为被ollvm混淆过，体积非常大，可能是fla和bcf都加上去了
这个函数IDA识别基本上是卡死状态，所以只能是找关键点切入
看一下OLLVM的图，被混的 单个switch有几千个，而且F5卡死了



所以用快速定位关键汇编位置

分析ollvm个人觉得一点技巧是找到关键的block，下好断点，走一遍，逆向查找，基本上如果不是很大的代码块都能梳理清楚逻辑
大致如下图

现在反方向去找到是从哪里获取到的字符串，这个字符串是如何从DexCode取出来的，那么这个vm解释执行的逻辑差不多就清楚了

倒推代码来了解逻辑

上面的onCreate是根据在函数j__S0l0$SOOII$0llllSI_O0S0ll__ll_S5lll5lOlI5SO0S5$ 这里，根据一个输入值返回的结构体来得到的

计算处一个全局变量的偏移值

return *(_QWORD *)(qword_7517D666A0 + 8LL * a1);

其实是个结构体

用IDA直接取字符串看一下

idc.GetString(idc.Qword(idc.Qword(idc.Qword(0x7517D666A0) + 0x5C * 1) + 8 *n))

n=0是类名　android/support/v4/app/FragmentActivity

n=1是方法的参数签名 (Landroid/os/Bundle;)V

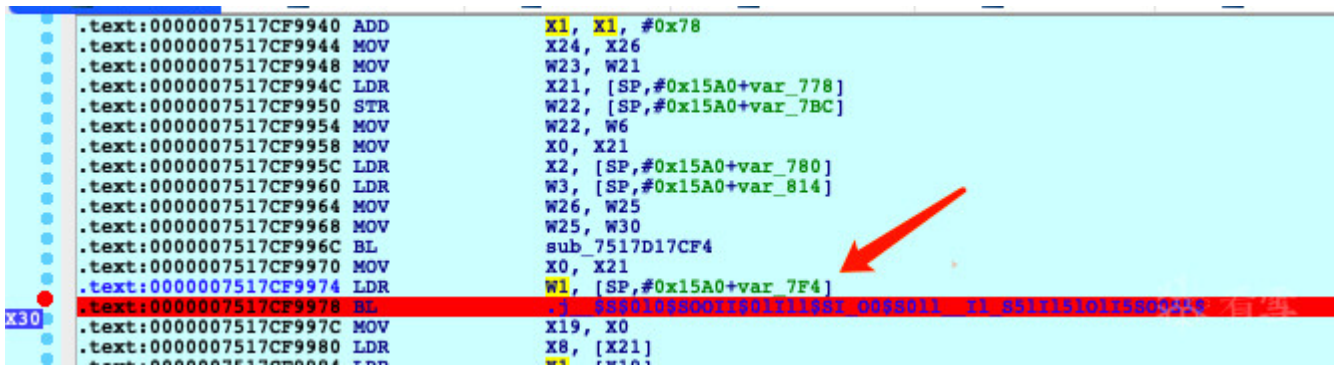
n=2是方法名称 onCreate

看起来是个如下的结构结构

```
struct {
    void *class_name;
    void *method_sig;
    void *method_name;
}
```

所以JNI调用的onCreate来自这个结构体，实际上如果做过java2c的一看就知道是调用super.onCreate在

然后再网上查看汇编，找到这个结构体是从哪里来的



函数入参存放在x1寄存器 就是w1，而且是在站栈上

LDR　　W1, [SP,#0x15A0+var_7F4]

根据这局汇编反向推一下　　LDR取值必然有一个STR赋值

STR　　X1, [SP,#0x15A0+var_7F4]

借助IDAPython来查找一下，之所以不用快捷键x去直接找，是因为需要找到调用顺序，所以在2b970的位置开始用脚本

```
last_insns = ''

def fn_f8():
    idaapi.step_over()
    GetDebuggerEvent(WFNE_SUSP | WFNE_CONT, -1)

def fn_f9():
    idaapi.continue_process()
    GetDebuggerEvent(WFNE_SUSP | WFNE_CONT, -1)


last_ins = ''
def run_next():
    fn_f8()

    asm_str = idc.GetDisasm(idc.GetRegValue('pc'))

    cur_match = re.match(r'STR\s+(\S+),\s\[SP,#0x15A0\+var_1460\]', asm_str,  re.M | re.I)
    if cur_match  :
        reg1 = cur_match.group(1)

        value = hex(idc.Word(idc.Qword(reg1) + 2))
        print('nop addr', hex(cur_addr), asm_str)
        return
    else:
        last_ins = asm_str

    run_next()


run_next()
```

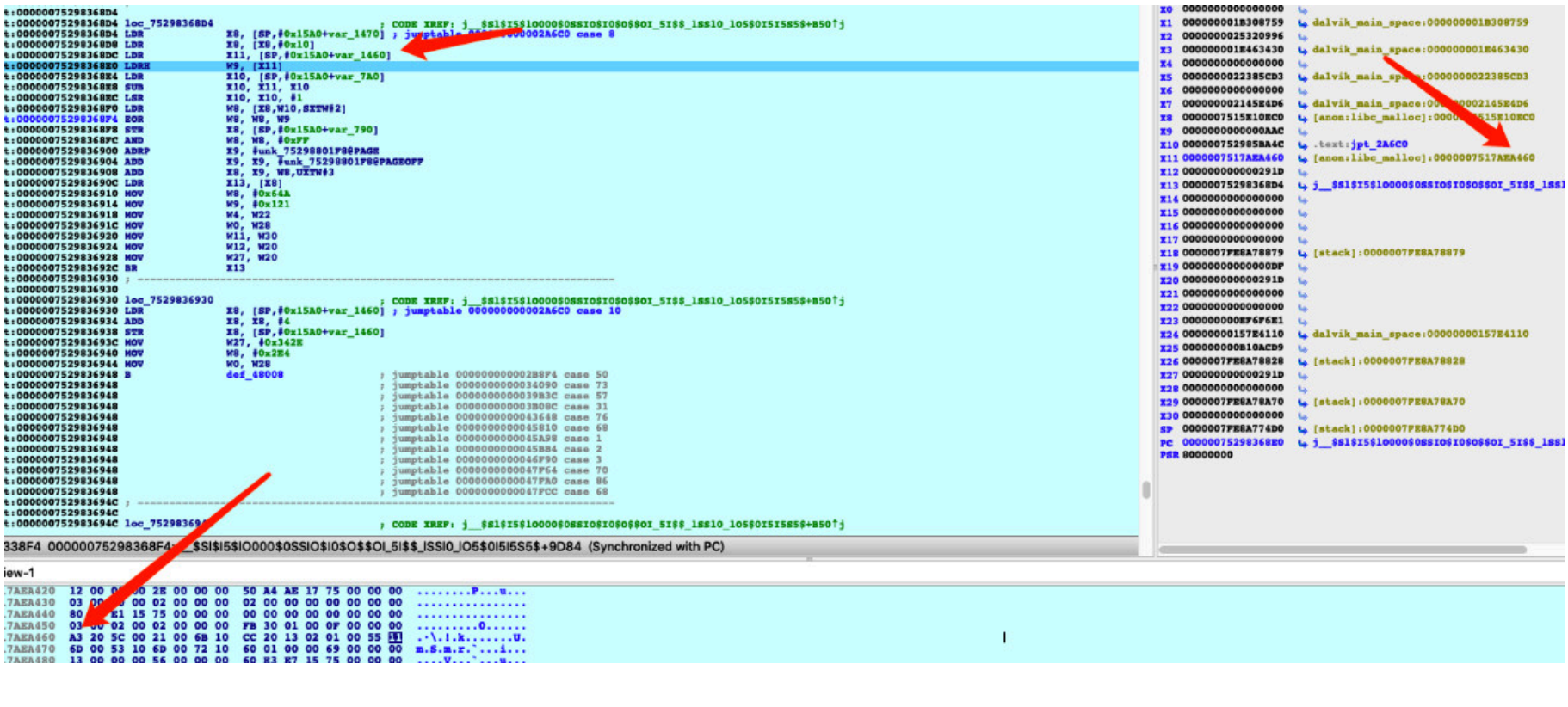
最终找到这个上面输入的5C是从最开始的 结构体里面的DexCode取出来的

如下

A3 20 5C 00 21 00

然后利用这个思路，找到指向指令insns的指针，实际就是在29b70处判断，当前的STR放入的指针是否是前边解出来的insns地址

找到指令指针是在存放在栈上的一个地址　[SP,#0x15A0+var_1460]



看上图所示，从栈 SP,#0x15A0+var_1460的地址 07517AEA460 得到的正是insns的地址

A3 20 5C 00 21 00

此时联系到onCreate的地方，用到了5c 而我们根据1460推导出了5c的来源

这里很清楚了，实际上指令的格式还是没变 则，解释执行OPCODE

还是6字节

A3 20 5C 00 21 00

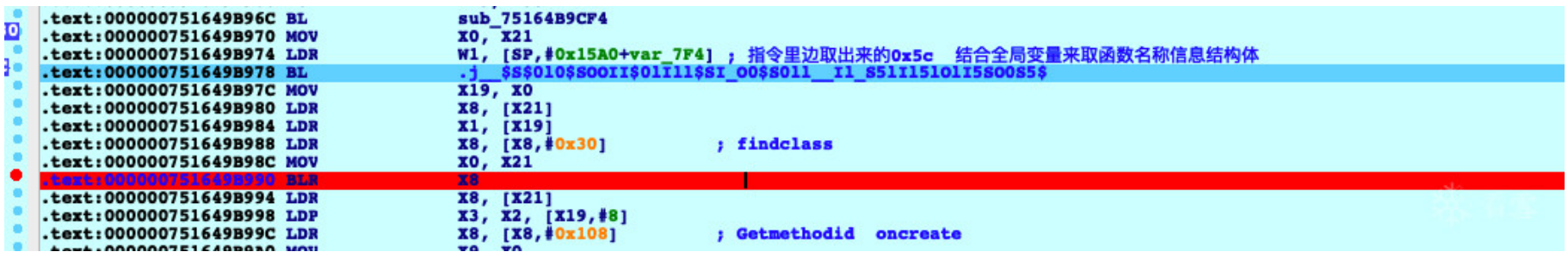
A | G | op BBBB F|E|D|C

对应一下就是 A=2 G=0 op=A3 BBBB是0x5C C=0 D=0 E=0 F=1

这里的第三组也就是 0021解密一下 0021 & 0xf = 0001

0x5c在Android自带虚拟机里变解释执行为 MethodID，这里vmp使用的是自定义存放的一个结构体，估计是为了快速查找，因为按照逻辑，是要从DEX里边取查找，可能是为了提高效率，所以保存起来

并且我看到vmp虚拟化的java函数越多，libdexjni.so的体积越大



继续调试往下走，你会看到 CallNonVirtualMethod 正是 super.onCreate

很熟悉的格式 [A=2] op {vC, vD}, kind@BBBB

---

a320 是invoke-super

005c是取MethodID

0021 解密0001 实际上是参数v0 但是我觉得这个解密多余的 因为取前边2位的

则这条指令是 invoke-super {p0, p1}, Landroidx/fragment/app/FragmentActivity;->onCreate(Landroid/os/Bundle;)V

```
L6494D88 LDR             X8, [SP,#0x15A0+var_1460] ; jumptable 00000000002A6CC0 case 674
L6494D8C ADD             X8, X8, #6                ; 往后6个字节，就是invoke-super解析完了的位置
L6494D90 STR             X8, [SP,#0x15A0+var_1460]
L6494D94 MOV             W8, #0x456
L6494D98 B               loc_75164B49C0
```

这里取出来是

6B 10

调试发现实际就是 定义了一个数值

6b const

10 v0,0x1

结果就是 const v0,0x1

---

next

CC 20 13 02 01 00[A=2]op{vC, vD},kind@BBBB

CC 20 invoke-virtual

0213 取MethodiD requestWindowFeature (I)Z

0001 参数

invoke-virtual {p0, v0}, Lcom/abing/appvmp/BaseActivity;->requestWindowFeature(I)Z

---

next

5511 6d00[A=2]op{vC, vD},kind@BBBB

1155 invoke-virtual

006d 取MethodiD requestWindowFeature (I)Z

0001 参数

iput-object p0, p0,Lcom/wangzhong/fortune/ui/activity/BaseActivity;->a:Lcom/wangzhong/fortune/ui/activity/BaseActivity;;

---

next 继续往下走，

在5feac处找到 这三句代码，运气不错，这里刻意F5，可以看到是 取出一个对象的值，根据分析得知是 BaseActivity的属性a

```
parseStringInfo(v8, v6[2]);              // 解密一下字符串  BaseActivity
v10 = 15 - v11;
v9 = 28501 - v11;
break;
case 1:
v12 = v9 - 151 * ((unsigned int)(55554 * v9) >> 23);
v3 = ((__int64 (__fastcall *)(JNIEnv *, __int64, __int64))(*v8)->GetObjectField)(v8, v7, v4);// 获取BaseActivity的a对象
v10 = 102 - v12;
if ( !v5 )
{
  v5 = 0LL;
  v10 = 103 - v12;
}
break;
case 2:
v13 = v9 - 11 * ((unsigned int)(47663 * v9) >> 19);
v5 = ((__int64 (__fastcall *)(JNIEnv *, _QWORD))(*v8)->FindClass)(v8, *v6);// FieldID获取
v4 = ((__int64 (__fastcall *)(JNIEnv *, __int64, _QWORD, _QWORD))(*v8)->GetFieldID)(v8, v5, v6[2], v6[1]);
if ( v4 )
```

合起来就是执行了 5310 6d00 这条指令

---

53 10 6D 00 [A=2]op{vC, vD},kind@BBBB

1053 invoke-virtual

006d 取MethodiD requestWindowFeature (I)Z

0001 参数

iget-object p0, p0,Lcom/wangzhong/fortune/ui/activity/BaseActivity;->a:Lcom/wangzhong/fortune/ui/activity/BaseActivity;

---

next 脚本执行结果如下

```
Python>idc.GetString(idc.Qword(idc.Qword(idc.Qword(0x75165086A0) + 0x160 * 8) + 8 *1))
(Landroid/app/Activity;)V
Python>idc.GetString(idc.Qword(idc.Qword(idc.Qword(0x75165086A0) + 0x160 * 8) + 8 *0))
```

---

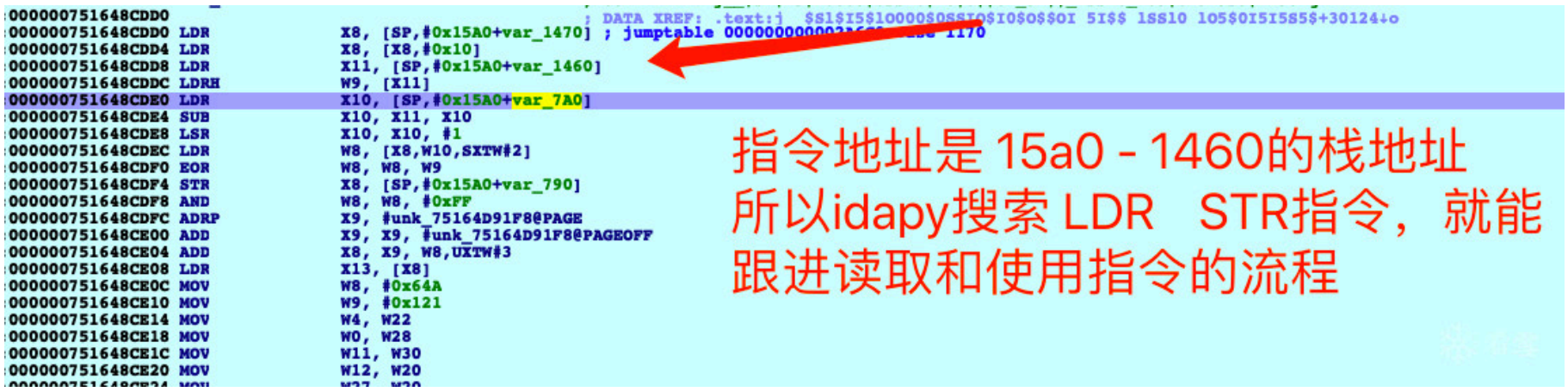
72 10 60 01 00 00 [A=2]op{vC, vD},kind@BBBB

10 72 invoke-virtual

0160 取MethodiD requestWindowFeature (I)Z

0000 参数编号

invoke-static {v0}, Lcom/wangzhong/fortune/f/c;->a(Landroid/app/Activity;)V



next

---

69 00 这个指令比较简单就是

return-void

对应到dex指令 ,0x5c这些部分需要自己取dex里边查找MethodID和ClassName对应起来，就是算出MethodID的索引就行

这里的5c最终是要到dex里取查找的

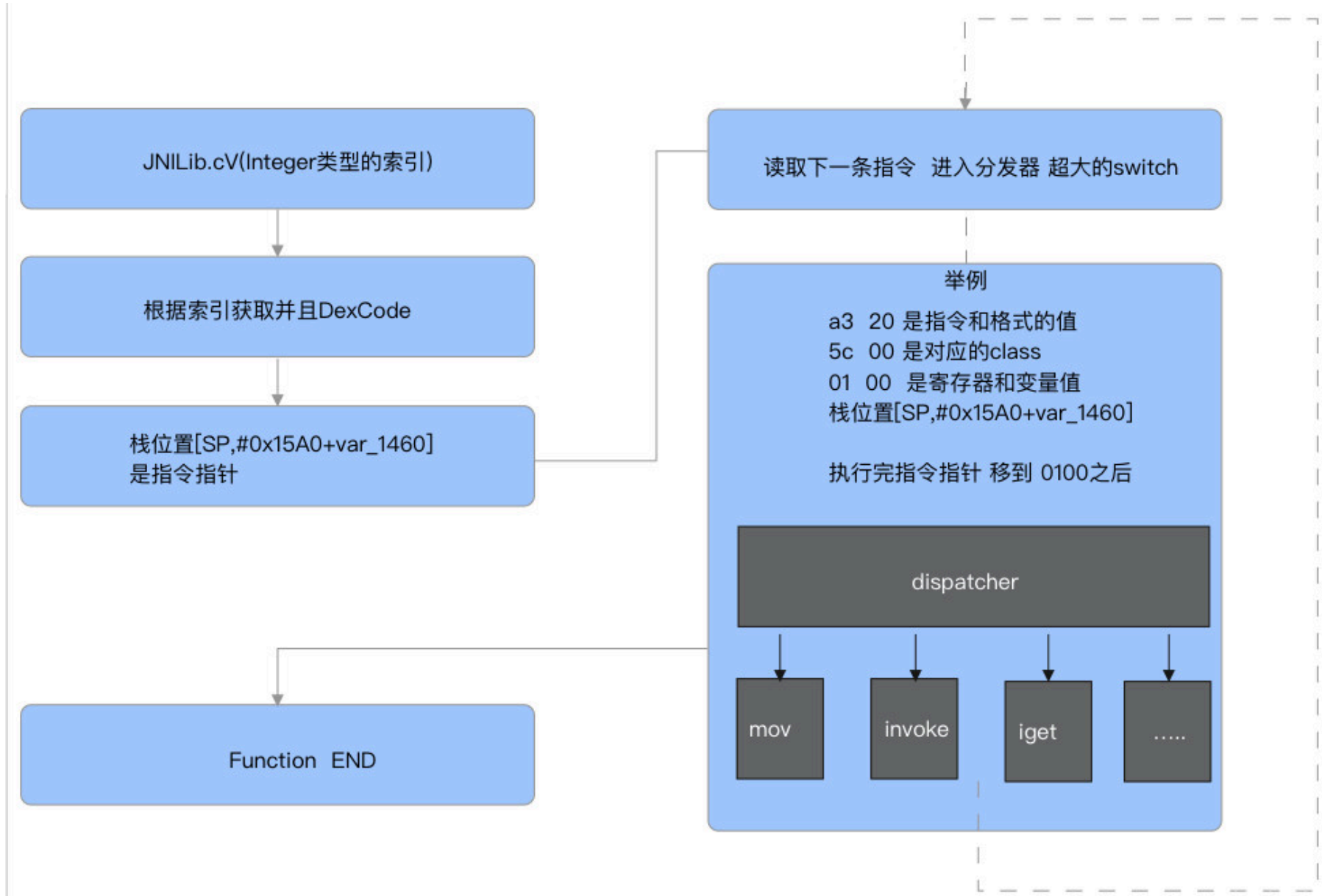
把下面这部分指令的根据分析经过转换

A3 20 5C 00 01 00 6B 10 CC 20 13 02 01 00 55 11

6D 00 53 10 6D 00 72 10 60 01 00 00 69 00 00 00

VMP的opcode		真是的Dex指令
A3 20 5C 00 01 00	------>	6f 20 02 15 21 00
6B 10	------>	12 10
CC 20 13 02 01 00	------>	6e 20 96 b1 01 00
55 11 6D 00	------>	5b 11 80 69
53 10 6D 00	------>	54 10 80 69
72 10 60 01 00 00	------>	71 10 ad ad 00 00
69 00	------>	0e 00

用流程图来说明下



得到

修复前的指令 实际上 JNILib.cv这部分代码是填充的 只有一个索引有用，所以直接覆盖

uint insns_size	16h
ushort insns[22]	
ushort insns[0]	3012h
ushort insns[1]	23h
ushort insns[2]	1D9Ch
ushort insns[3]	112h
ushort insns[4]	44Dh
ushort insns[5]	100h
ushort insns[6]	1112h
ushort insns[7]	54Dh
ushort insns[8]	100h
ushort insns[9]	2112h
ushort insns[10]	313h
ushort insns[11]	12h
ushort insns[12]	1071h
ushort insns[13]	BD59h
ushort insns[14]	3h
ushort insns[15]	20Ch
ushort insns[16]	24Dh
ushort insns[17]	100h
ushort insns[18]	1071h
ushort insns[19]	6099h
ushort insns[20]	0h
ushort insns[21]	73h

修复后的指令

ushort outs_size	1h
ushort tries_size	0h
uint debug_info_off	0h
uint insns_size	16h
▼ ushort insns[22]	
ushort insns[0]	206Fh
ushort insns[1]	1502h
ushort insns[2]	21h
ushort insns[3]	1012h
ushort insns[4]	206Eh
ushort insns[5]	B196h
ushort insns[6]	1h
ushort insns[7]	115Bh
ushort insns[8]	6980h
ushort insns[9]	1054h
ushort insns[10]	6980h
ushort insns[11]	1071h
ushort insns[12]	ADADh
ushort insns[13]	0h
ushort insns[14]	0h
ushort insns[15]	0h
ushort insns[16]	0h
ushort insns[17]	0h
ushort insns[18]	0h
ushort insns[19]	0h
ushort insns[20]	0h
ushort insns[21]	Eh

实际指令是 0xF所以其他的nop掉 最后给一个return void 就可以了

这里比较坑的一点是寄存器的数量一定要改，不然的话dex2jar转不了

修复前

```
    throw new VerifyError("bad dex opcode");
  }

  protected void onCreate(Bundle paramBundle)
  {
    JniLib.cV(new Object[] { this, paramBundle, Integer.valueOf(18) });
    throw new VerifyError("bad dex opcode");
  }
```

修复后

```
  public void o()
  {
    JniLib.cV(new Object[] { this, Integer.valueOf(17) });
    throw new VerifyError("bad dex opcode");
  }

  protected void onCreate(Bundle paramBundle)
  {
    super.onCreate(paramBundle);
    requestWindowFeature(1);
    this.a = this;
    c.a(this.a);
  }

  protected void onDestroy()
  {
    JniLib.cV(new Object[] { this, Integer.valueOf(19) });
    throw new VerifyError("bad dex opcode");
  }

  protected void onPause()
```

总结：

1-是用JNI来解释执行opcde的

2-op被替换了，但是 A G 那部分参数寄存器数字是不会变的，因为vmp也需要指定是几个参数，来使用

4-这里的op可能被加密了，个人愚见人为这个Op加密不加密无所谓，因为最终实际上是个对应关系 0xff个opcode对应0xff个opcode

hookjni 可以看到很多输出信息 就是说vmp实际采用的还是 jni来实现

如果要全部都替换掉，需要挨个分析指令，做一个映射表出来

目前来看还是java2c + arm指令虚拟化应该是比较保险的操作，因为自己写一个解释器，纯自己实现指令，肯定问题非常多，所以指令还是通过Jni来实现的，
但是效率貌似低了些，如果这种方式加上ARM指令虚拟化，分析起来可就难受很多了

样本是以前的版本，目的是为了分析和学习，这里只提供so文件，交流经验，需要样本私聊我

最后于 2020-1-21 11:13 被贝a塔编辑 ，原因:

上传的附件:

libdexjni.so（627.17kb，156次下载）

赞赏记录

参与人	雪币	留言	时间
sinker_		期待更多优质内容的分享，论坛有你更精彩！	2025-3-15 10:10
PLEBFE		为你点赞~	2022-7-27 01:29
心游尘世外		为你点赞~	2022-7-26 23:19
飘零丶		为你点赞~	2022-7-17 02:48
黑的默		为你点赞~	2020-7-30 11:23
CDD..		为你点赞~	2020-6-22 11:54
0x指纹		为你点赞~	2020-6-19 18:52
xhyeax		为你点赞~	2020-6-2 13:10
飞翔的牧人		为你点赞~	2020-2-1 23:18
Editor		为你点赞~	2020-1-10 10:04
zpob		为你点赞~	2020-1-8 15:32
V1NKe		为你点赞~	2020-1-7 10:05

wx_益达	为你点赞~	2020-1-7 09:44
bluth	为你点赞~	2020-1-6 15:29
贝a塔	为你点赞~	2020-1-6 15:28
jmpcall	为你点赞~	2020-1-6 12:48

最新回复 (14)

yezheyu 7

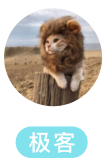
2 楼

极客

先收藏哈哈

2020-1-5 06:49

0 •••



yezheyu 7

3 楼

极客

感谢分享，膜拜🙏

2020-1-5 07:03

0 •••

Francissss 2

4 楼

极客

收藏。

2020-1-5 10:11

0 •••

不知世事 7 1

5 楼

大侠

java2c + arm指令虚拟化相对于dexVMP要牺牲体积和性能

2020-1-5 11:14

0 •••

bluth 5

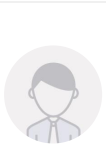
6 楼

极客

bangbang的企业级吧，貌似棍棍的只看到这篇文章

2020-1-6 11:34

0 •••



壹久玖 3

7 楼

极客

m

2020-1-6 17:18

0 •••

Editor 16

9 楼

版主

感谢分享!

2020-1-10 10:04

0 •••

BOSSSUN 9

10 楼

极客

求样本😳

2020-1-10 22:03

0 •••

首页 社区 课程 招聘 发现

最新回复 (14)

浅笑不语 9

11 楼

极客

收藏了

2020-6-10 21:16

0 •••

caolinkai 9

12 楼

极客

牛了。

2020-6-18 23:05

0 •••

it爱好者_625041 6

13 楼

极客

求样本

2020-7-12 22:54

0 •••

mb_whnfigai 0

14 楼

临时

请问指令格式里面有形如kind@BBBB的，最后是如何确定这个kind的呀?

2020-8-1 20:33

0 •••

mb_gbrkqlle 0

15 楼

临时

大神求样本，等级不能私信，看到请回，摆脱

2022-4-12 20:16

0 •••

贝a塔

3 年没回复的帖子已锁定，有问题请联系版主!

回帖 表情 高级回复

返回