需求

1.1 需求概述

汇编语言是一类功能强大的程序设计语言，也是利用计算机所有硬件特性并能直接控制硬件的语言。目前在嵌入式开发、单片机开发、系统软件设计、某些快速处理、位处理、访问硬件设备等高效程序的设计方面有较多应用。而 ARM 处理器是一种 16/32 位的高性能、低成本、低功耗的嵌入式 RISC 微处理器，由 ARM 公司设计，目前已经成为应用最为广泛的嵌入式处理器。

Hook 意为钩子，勾住系统的程序逻辑和执行流程，同时嵌入在源码某段逻辑执行的过程中，通过代码手段拦截执行该逻辑，加入自己的代码逻辑。

本次课程期末考核报告基于 ARM 汇编语言实现 Inline Hook。主要在学习 ARM 汇编语言的过程中掌握 ARM 指令集的使用，熟悉 ARM 处理器的基本架构，并且通过 Android Studio 中的 Android Emulator 虚拟环境，结合 IDA 反汇编工具进行调试分析，最终编码实现了作为 Hook 技术之一的 Inline Hook。

其中 Inline Hook 的核心思想是：通过替换 Hook 地址原始的指令，实现在指令执行之前跳转到 Trampoline 区域，执行完毕跳转回到 Original 区域继续执行，跳转到的 Trampoline 区域通常为自己编写的 Proxy 函数。Inline Hook 技术的实现对于编写补丁、动态修正、获取上下文信息有着重大意义，如图 1 所示为 Inline Hook 内核函数执行流程图。

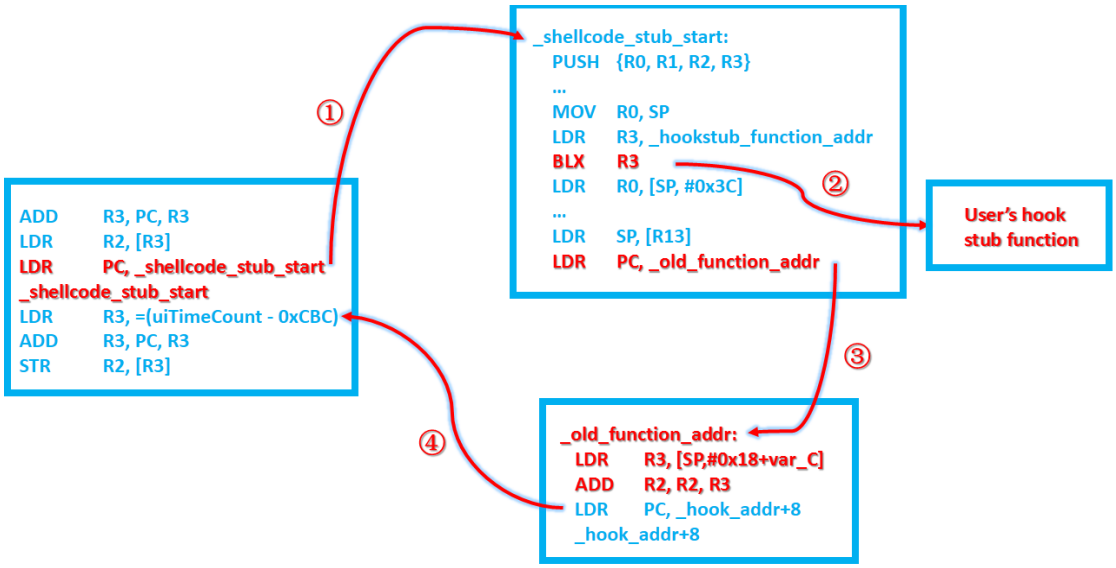


图 1 Inline Hook 函数执行流程图

1.2 开发环境

1. 软件环境：
   - Windows 10 Build 1909 操作系统；
   - Android Studio（模拟器：32 位 ARM （armeabi-v7a）、NDK 交叉编译链、Android Tools 工具包）；
   - IDA（远程调试工具，需要安装 Keypatch 插件、Capstone、IDA Python）；
   - Visual Studio Code（编辑器、集成控制台）。
2. 硬件环境：PC 端：处理器 Intel(R) Core(TM) i9-10900K CPU @ 3.70GHz。
3. 参考：
   - Arm Architecture Reference Manual 手册
   - man 手册

1.3 启动环境

为方便执行，需要启动的环境条件都写为了.bat 文件。

1. 启动模拟器（start_emu.bat）
2. 启动 android_server
   - push android_server

```
C:\Users\Kaola>adb push F:\Tools\IDA_7.0\dbgsrv\android_server /data/user
F:\Tools\IDA_7.0\dbgsrv\android_server: 1 file pushed, 0 skipped. 74.7 MB/s (589588 bytes in 0.008s)
```

   - 更改权限

```
C:\Users\Kaola>adb shell
root@android:/ # cd /data/user
root@android:/data/user # chmod 777 android_server
```

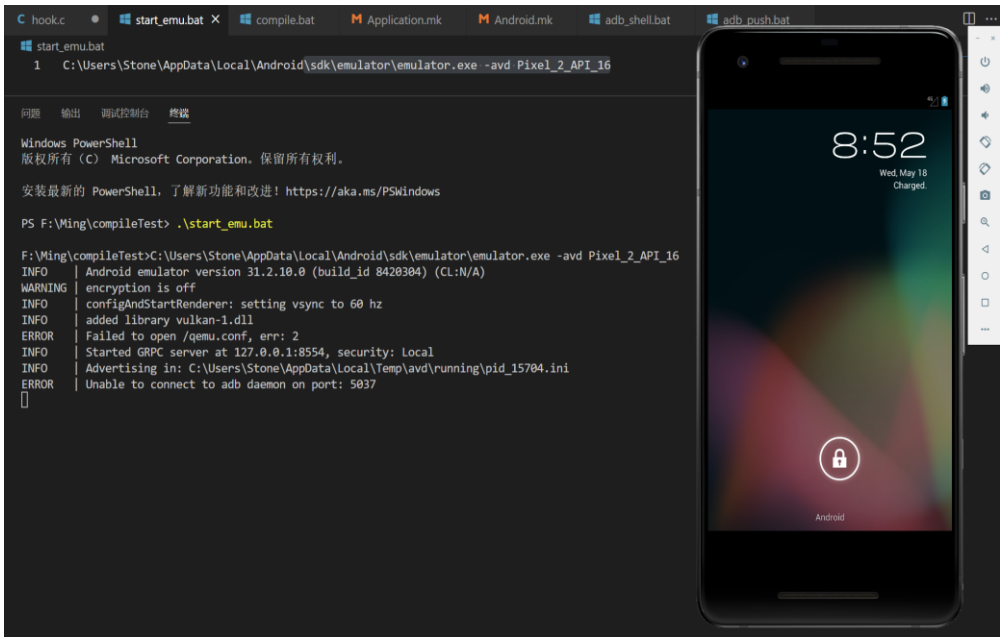
   - 启动 android_server（./android_server &(关闭 shell 进程仍在)

```
1|root@android:/data/user # ./android_server &
[1] 2042
root@android:/data/user # IDA Android 32-bit remote debug server(ST) v1.22. Hex-Rays (c) 2004-2017
Listening on 0.0.0.0:23946...
```

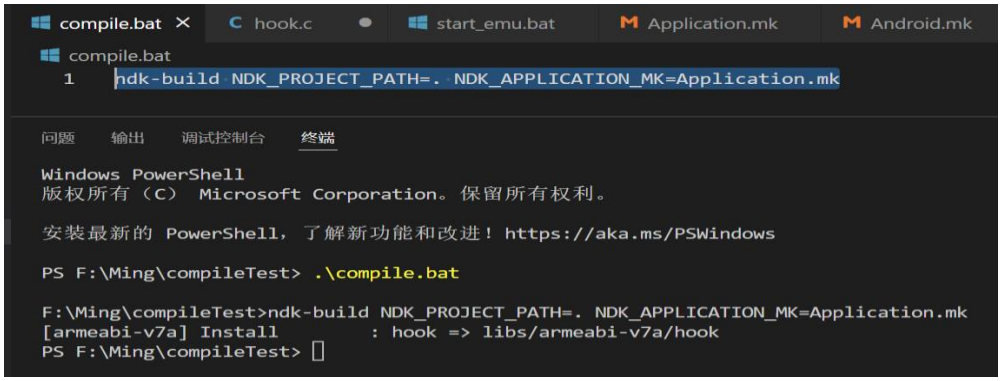
   - 端口转发

```
C:\Users\Kaola>adb forward tcp:23946 tcp:23946
23946
```

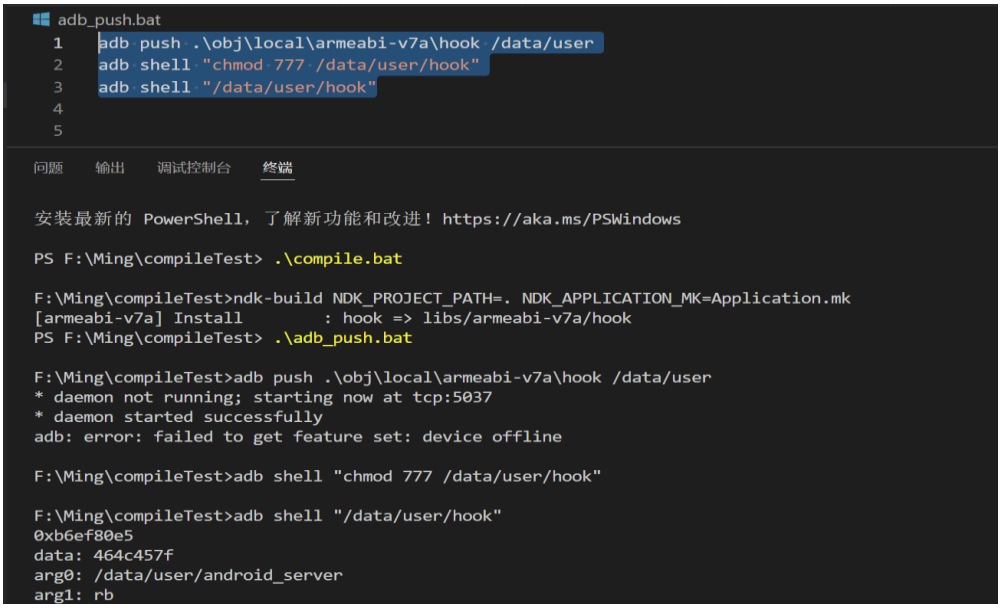
3. 执行 obj 目录下的 hook 文件流程
   - start_emu.bat



- compile.bat



- adb_push.bat



- ./android_server &

```
PS F:\Ming\compileTest> adb shell
root@android:/ # cd /data/user
root@android:/data/user # ./android_server &
[1] 1874
root@android:/data/user # IDA Android 32-bit remote debug server(ST) v1.22. Hex-Rays (c) 2004-2017
Listening on 0.0.0.0:23946...
```

- Adb_shell.bat

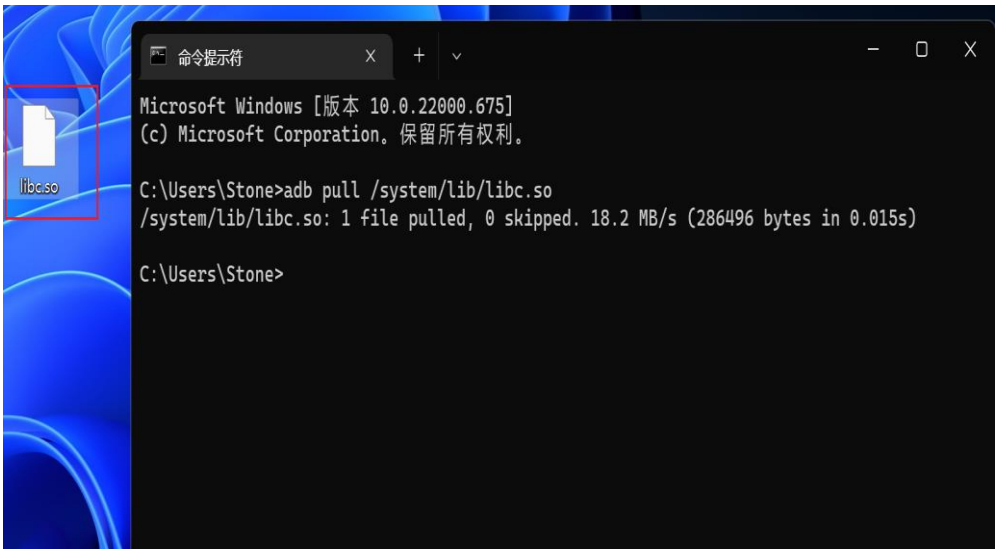
```
PS F:\Ming\compileTest> .\adb_shell.bat

F:\Ming\compileTest>adb forward tcp:23946 tcp:23946
23946

F:\Ming\compileTest>adb shell
root@android:/ #
```

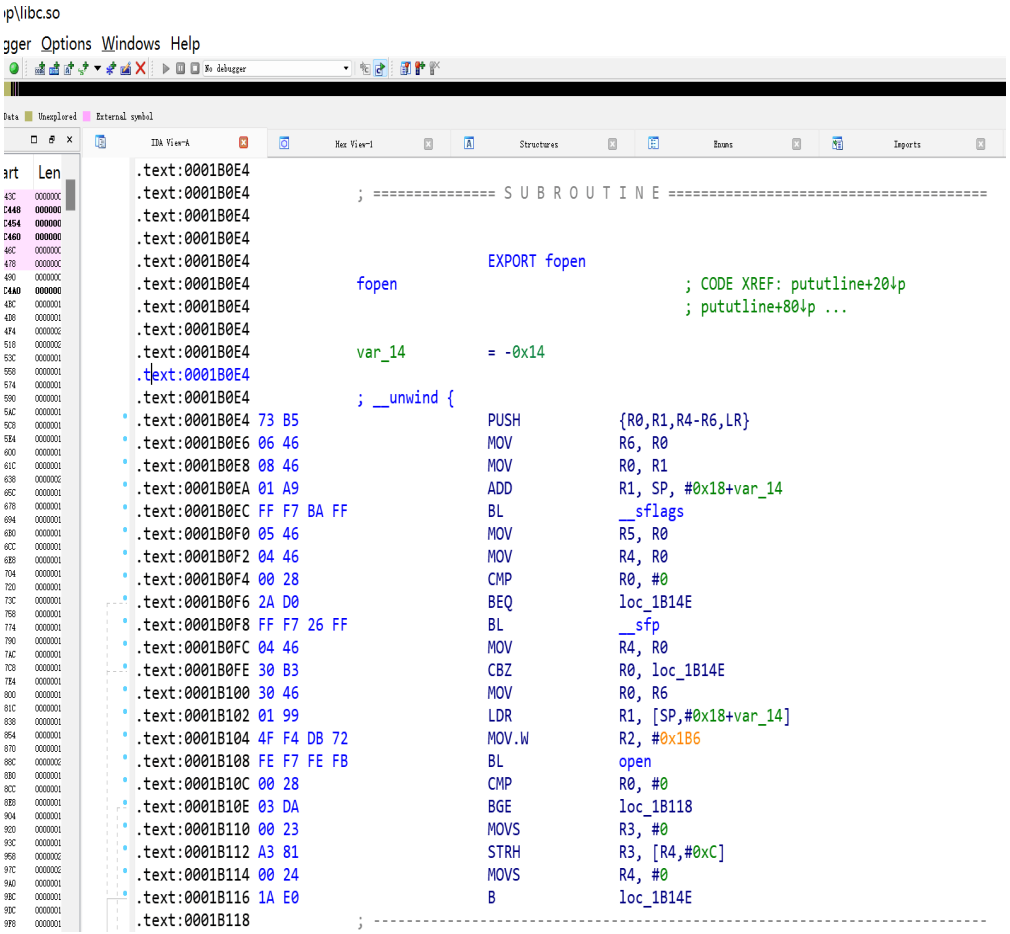
- IDA attach hook 进行调试

```
1653        [32] com.android.quicksearchbox
1837        [32] /system/bin/sh -c /data/user/hook
1839        [32] /data/user/hook
1868        [32] /system/bin/sh -
```

- 获取 libc.so

```
PS F:\Ming\compileTest> adb shell
root@android:/ # cat /proc/1839/maps
b6ebd000-b6ebe000 r--p 00000000 00:00 0
b6ebe000-b6ec6000 r--s 00000000 00:0a 48         /dev/__properties__ (deleted)
b6ec6000-b6edb000 r-xp 00000000 fe:00 668        /system/lib/libm.so
b6edb000-b6edc000 r--p 00014000 fe:00 668        /system/lib/libm.so
b6edc000-b6edd000 rw-p 00000000 00:00 0
b6edd000-b6ef8000 r-xp 00000000 fe:00 605        /system/lib/libc.so
b6ef8000-b6ef9000 rwxp 0001b000 fe:00 605        /system/lib/libc.so
b6ef9000-b6f20000 r-xp 0001c000 fe:00 605        /system/lib/libc.so
b6f20000-b6f23000 rw-p 00043000 fe:00 605        /system/lib/libc.so
b6f23000-b6f2e000 rw-p 00000000 00:00 0
b6f2e000-b6f2f000 r-xp 00000000 fe:00 719        /system/lib/libstdc++.so
```





2 ARM 汇编

2.1 寄存器与指令基本格式

1. ARM 下的指令规则为：
   - 没有隐式内存操作指令
   - 拥有 0-3 个操作数，内存操作数和立即数操作数不能同时存在
   - 内存操作数至多出现一次，寄存器操作数总在最前面
2. ARM 总共有 37 个寄存器，其可以分为以下 2 类：
   - 通用寄存器（31 个）

     不分组寄存器（R0—R7）（8 个）

     分组寄存器（R8—R14）

     PC（R15）
   - 程序状态寄存器（6 个）

     CPSR（1 个）

SPSR（5 个）

3. ARM 指令基本包含以下 5 大类：
   - MOV 指令
   - 基本运算：ADD(ADR ADRL)，SUB，RSB，AND，BIC，ORR，EOR，LSL，LSR，ASR
   - 访存指令：LDR，STR
   - 块访存指令：LDMFD==LDMIA；STMFD==STMIA
   - 分支跳转：B，BL，BX，BLX

2.2 读 PC 寄存器

R15（PC）总是指向“正在取指”的指令，而不是指向“正在执行”的指令或正在“译码”的指令。一般来说习惯性约定将“正在执行的指令作为参考点”，称之为当前第一条指令，因此 PC 总是指向第三条指令。

在 ARM 状态时，每条指令为 4 字节长，所以 PC 始终指向该指令地址加 8 字节的地址，即：PC 值=当前程序执行位置+8。

2.3 条件和 NZCV 标志位响应

- SUBS 既要结果又要标志寄存器（标志位响应）
- SUB 只要结果忽略标志寄存器
- CMP 只要标志位忽略结果
- ADDS 既要结果又要标志寄存器（标志位响应）
- ADD 只要结果忽略标志寄存器
- CMN 只要标志位忽略结果

2.4 MOV 指令

MOV 指令中第一个操作数一定为寄存器（register），第二个可以是寄存器或者立即数（immediate）。

- MOV, MOVS (immediate)
- MOV, MOVS (register)
- MOV, MOVS(register-shifted register) 带有偏移的寄存器
- MOVT

2.5 基本整型运算

基本整型运算涉及到 3 个操作数：寄存器、立即数。

- 加法 ADD，其中 ADR 涉及到伪指令，CMN 影响标志位；
  如图 2 所示：R2 带 shifted 偏移 4

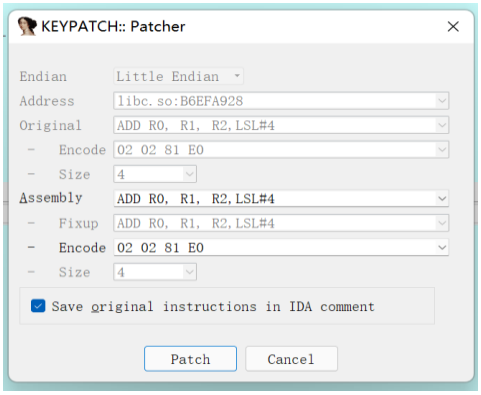


图 2　ADD 实例

- 减法 SUB，CMP 影响标志位，反减 RSB 第三个操作数减去第二个操作数给第一个操作数；
- 与 AND，TST 影响标志位；
- 反与 BIC，第二个操作数和第三个操作数取反再 And；
- 或 ORR，异或 EOR,TEQ 影响标志位；
- 移位指令：左移 LSL、逻辑右移 LSR、算数右移 ASR。

2.6 访存指令（LDR 读内存，STR 写内存）

1. 关注数据流向：（LDR-LOAD 加载内存），从内存向寄存器传递数据；（STR-STORE 存储）寄存器向内存传递数据。

2. 关注操作的寄存器和内存地址：第一个操作数为寄存器，第二个操作数带中括号为访问内存的方式，读取其地址中的内容（寄存器加上偏移）

```
libc.so:B6F8E928 04 00 91 E5      LDR           R0, [R1,#4]          ; Keypatch modified this from:
libc.so:B6F8E928                                                     ;   SVC 0
libc.so:B6F8E92C 02 02 91 E7      LDR           R0, [R1,R2,LSL#4]    ; Keypatch modified this from:
```

3. 关注后续附加行为

举例：

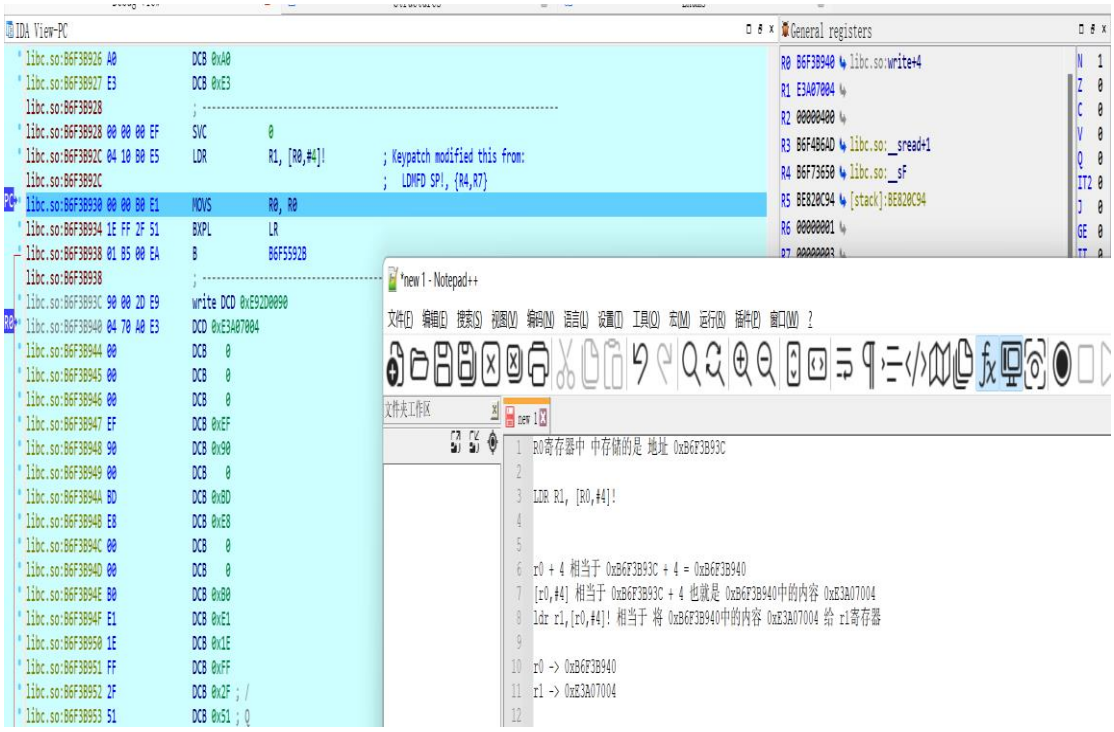
- LDR R1, [R0,#4]! 如下图例所示：
  R0 寄存器中存储的是地址 0xB6F3B93C；

R0 + 4 相当于 0xB6F3B93C + 4 = 0xB6F3B940；

[R0,#4]相当于地址 0xB6F3B940 中的内容 0xE3A07004；

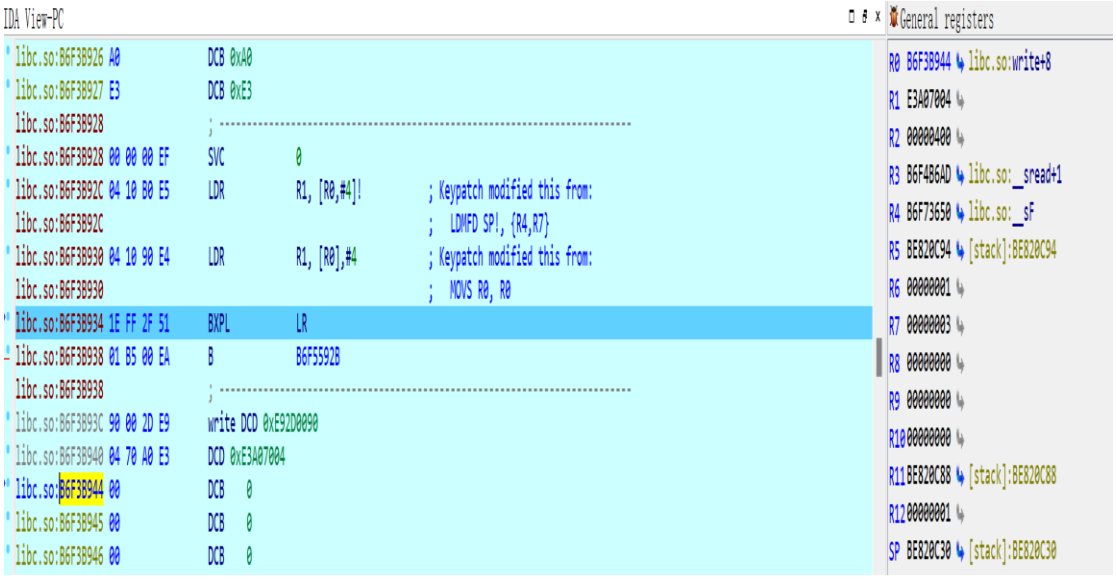
LDR R1, [R0,#4]!相当于将 0xB6F3B940 中的内容(内存)0xE3A07004 给 R1 寄存器，其中有！代表 R0+4 为 R0 新的值，也就是 0xB6F3B940；

此时 R1 的值为 0xE3A07004，而 R0 寄存器中存储的是地址 0XB6F3B940。

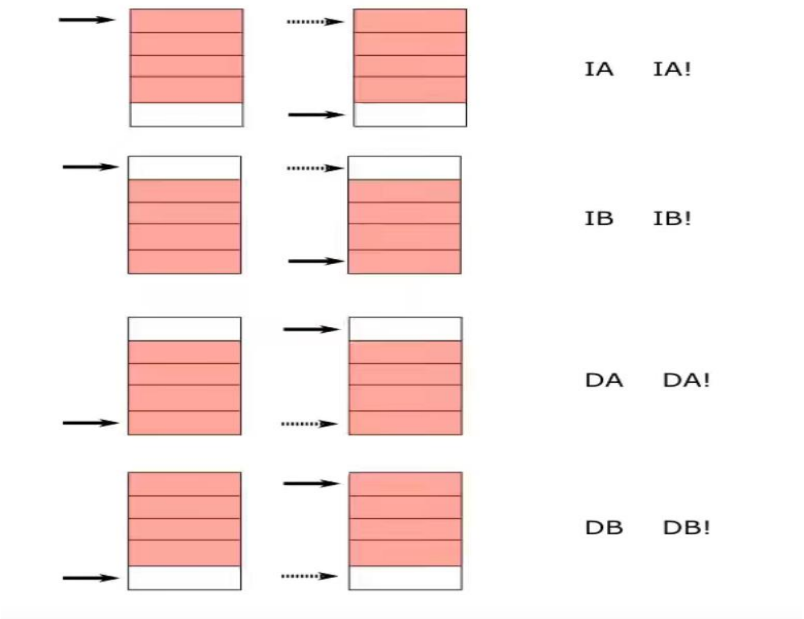


- LDR R1, [R0],#4

[R0]的内存 0xE3A07004 给 R1，再发生外偏移：R0 地址+4: B6F3B944



2.7 块访存指令

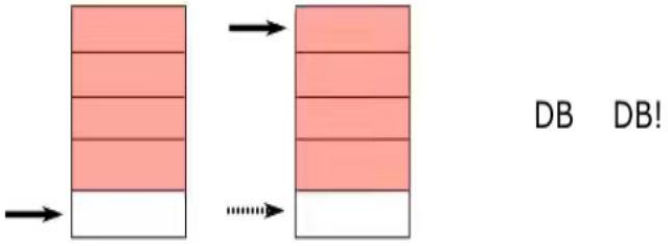


其中：

- LDMIA（load memory 读内存到寄存器，IA increase after）代表边读边增加 4 字节
- LDMIA R0,{R1-R4}低地址放到 R1，高地址放到 R4，从低到高（寄存器组）
- IB（increase before）先加 4 字节再读，与 IA 差了一个字节
- DA(decrease after)反向，边读边减 4 字节（向上移动指针）
- DB(decrease before)反向，先减 4 再读
- ！表示最后操作完指针是否跟随

■ 栈操作：LDMIA + STMIA 可以进行快速复制内存

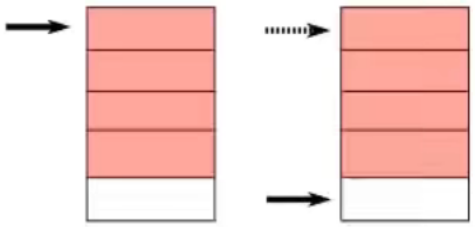
1. 入栈 push == STMDB(==STMFD) 存储数据，用到 SP 栈指针
   STMDB SP!，{r0-r3}相当于 STMFD SP!，{r0-r3}
   （DB 后为 SP 指针，IDA 会翻译成 FD）
   同时 R3,R2,R1,R0 依次存入栈（从右向左），此时 R3 在栈底



2. 出栈 Pop == LDMIA(==LDMFD)

LDMIA SP! ,{R1-R4}相当于 LDMFD SP! ,{R1-R4}

从左向右依次出栈 R1 R2 R3 R4，R1 在栈顶



IA IA!

2.8 分支跳转指令和模式切换

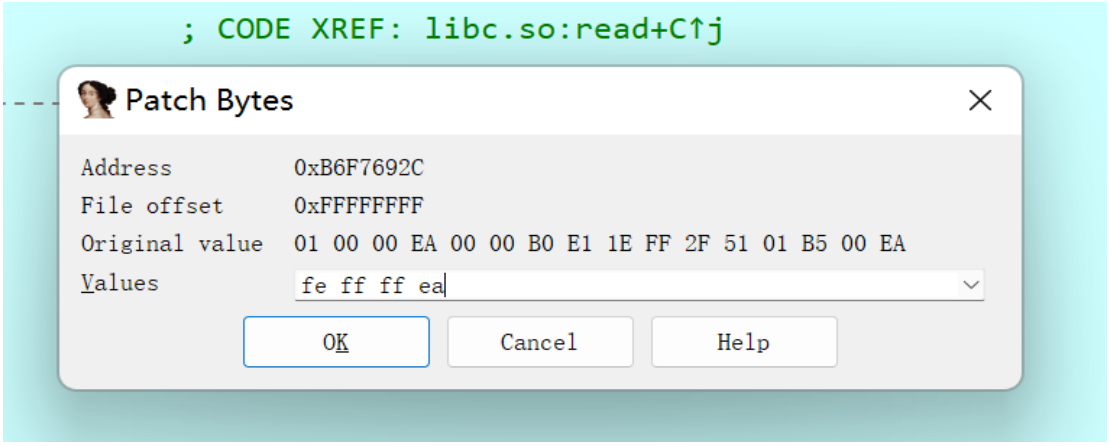
分支跳转指令包含：B、BL、BX、BLX，其中：

- B 和 BL 指令只能加立即数；
- BX 只能加寄存器；
- BLX（分两种不同的指令行为看）：既可以加立即数，也可以加寄存器。

1. B：无条件跳转，不带 link，不带模式切换。

```
libc.so:B6F76928 00 00 00 EF      SVC         0
libc.so:B6F7692C 00 00 00 EA      B           locret_B6F76934       ; Keypatch modified this from:
libc.so:B6F7692C                                                    ;   LDMFD SP!, {R4,R7}
libc.so:B6F76930                  ; ---------------------------------------------------------------------------
libc.so:B6F76930 00 00 B0 E1      MOVS        R0, R0
libc.so:B6F76934
libc.so:B6F76934                  locret_B6F76934                   ; CODE XREF: libc.so:read+C↑j
libc.so:B6F76934 1E FF 2F 51      BXPL        LR
libc.so:B6F76938 01 B5 00 EA      B           B6F9092B
libc.so:B6F76938                  ; ---------------------------------------------------------------------------
```

- 常用操作：修改 Patch Bytes：fe ff ff ea



- 因为每加 1 相当于×4 字节，fe ff ff 为-2，而 Pc+8-2*4 为跳转回自身。

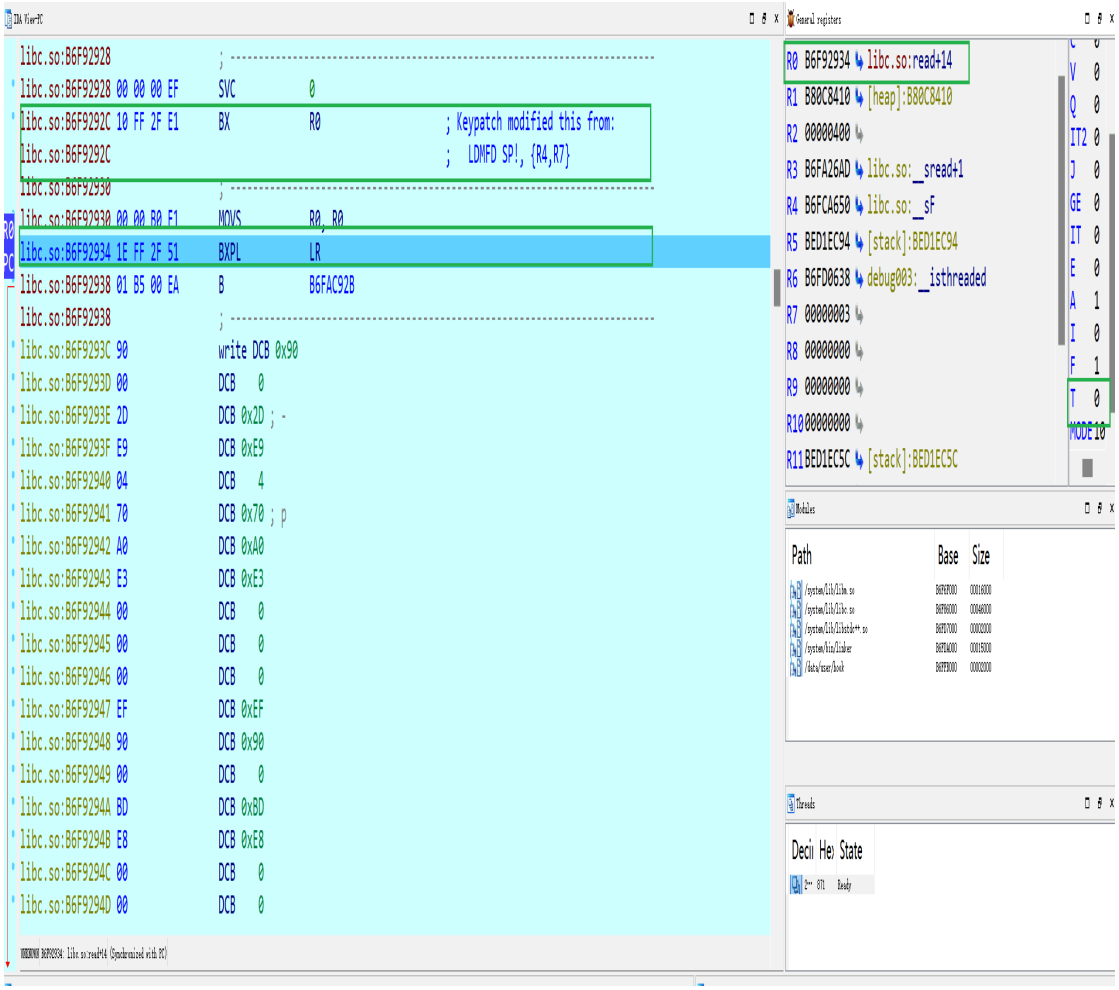
```
libc.so:B6F7692C FE FF FF EA        B               loc_B6F7692C
```

2. BL：不带模式切换，会写 LR 寄存器。BL 指令把当前指令行的下一个指令地址（要返回的地址）和当前 T 位（0 为 ARM,1 为 THUMB） 做或操作，给 LR 寄存器（根据 LR 的最低位，若为 0 返回到 ARM；若为 1，返回到 THUMB 模式）。

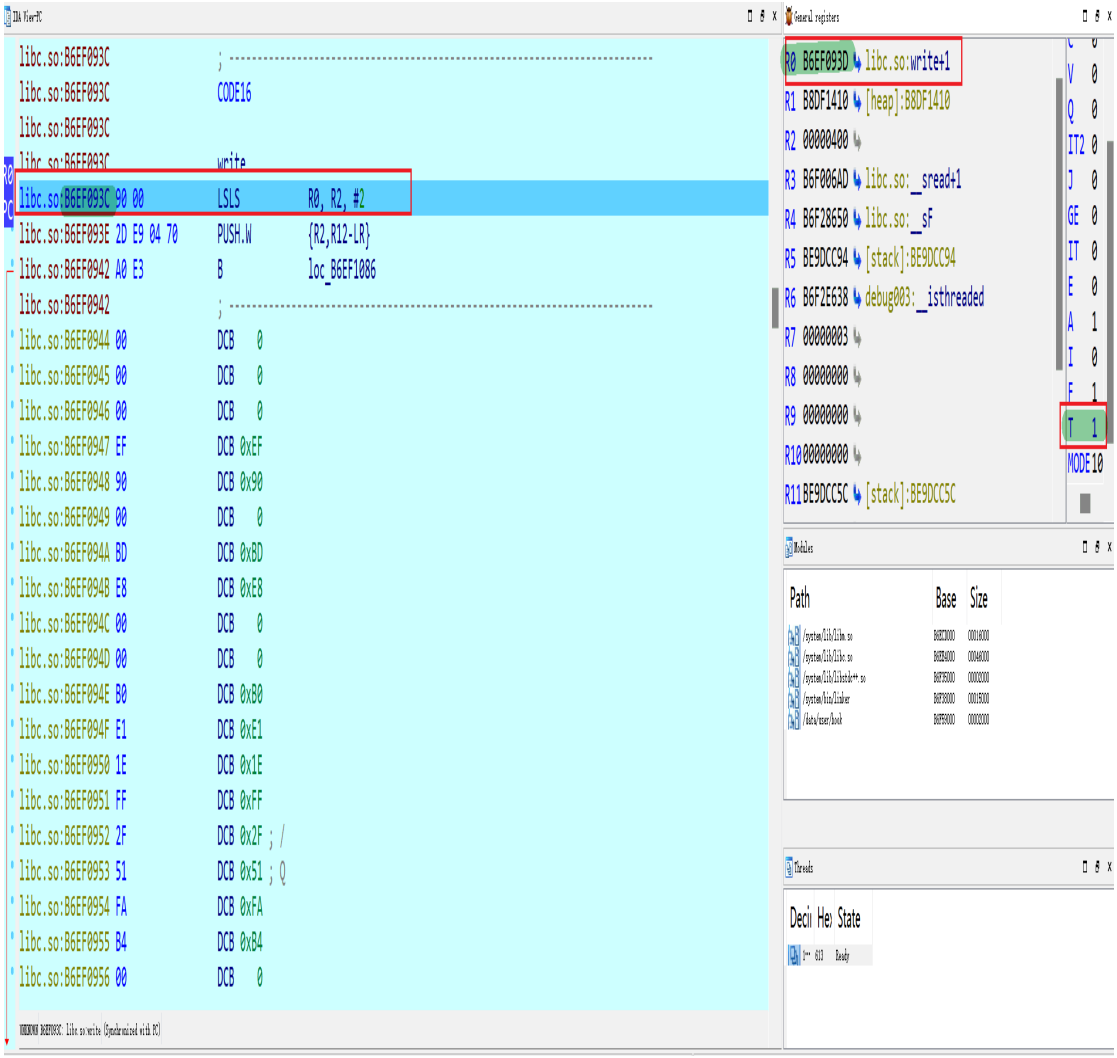
3. BX：只能加寄存器，同时带 X 不一定发生模式切换。BX 指令将寄存器的值拆成两部分，最低位写入 T 标志位，剩下的写入 PC 寄存器跳转过去。

■ 其中是否发生状态切换：需要根据 BX 跳转的寄存器地址，最低位写入想要跳转的新地址模式（ARM/THUMB）。

- 寄存器值直接更改为目标地址，不发生状态切换，T 标志位不变。



- 如果想要跳转到 ARM 模式，最低位写 0；如果想要跳到寄存器目标地址发生模式切换：如跳转到 0xB6F9493C 并发生模式切换。
  （T 位更改，从当前 arm 变为 thumb），将寄存器地址 R0 更改加 1，即 R0 值改为 0xB6F9493D，此时 T 标志位变为 1。



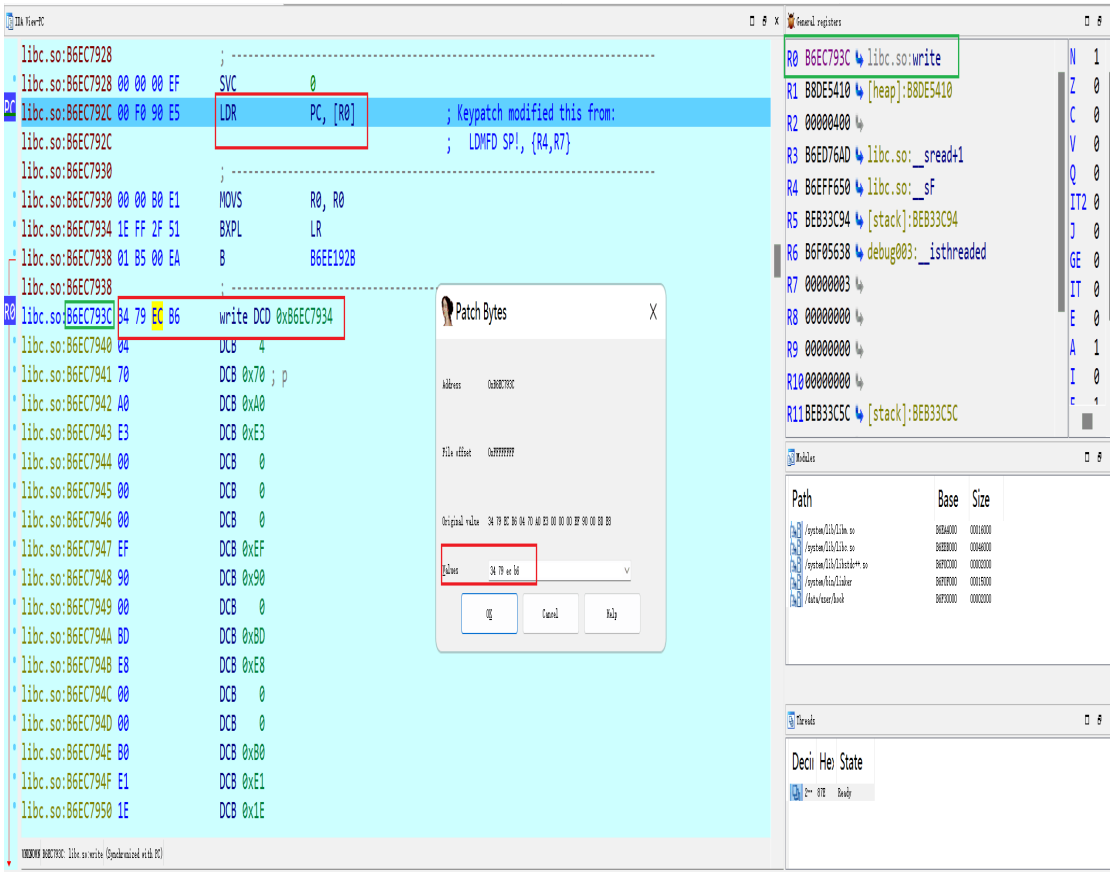
4. BLX：加上立即数，一定带有状态切换，并且会写 LR 寄存器。

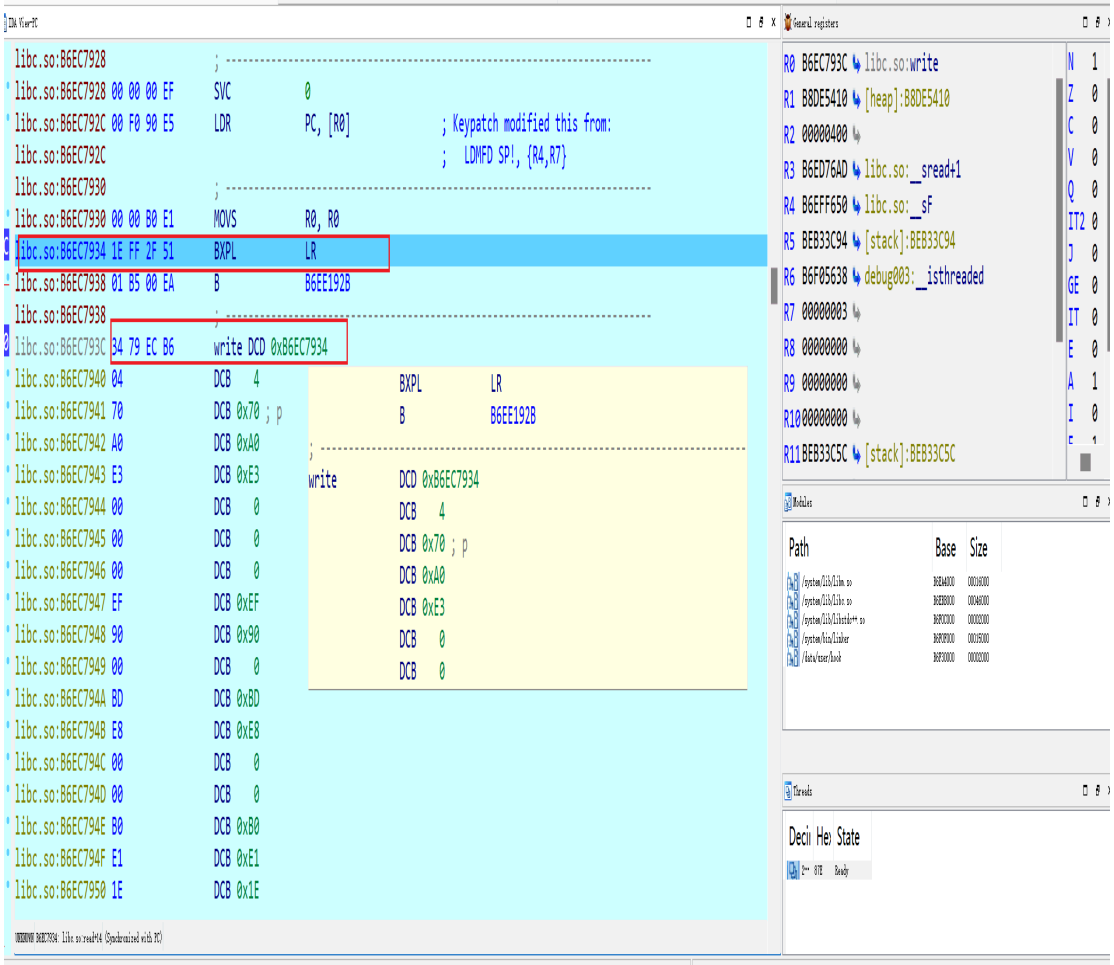
5. BLX：加上寄存器，此时 BLX 指令相当于 BX 指令（分情况分析），不一定带有状态切换，并且会写 LR 寄存器。

■ 附加：写 PC 的三种方式

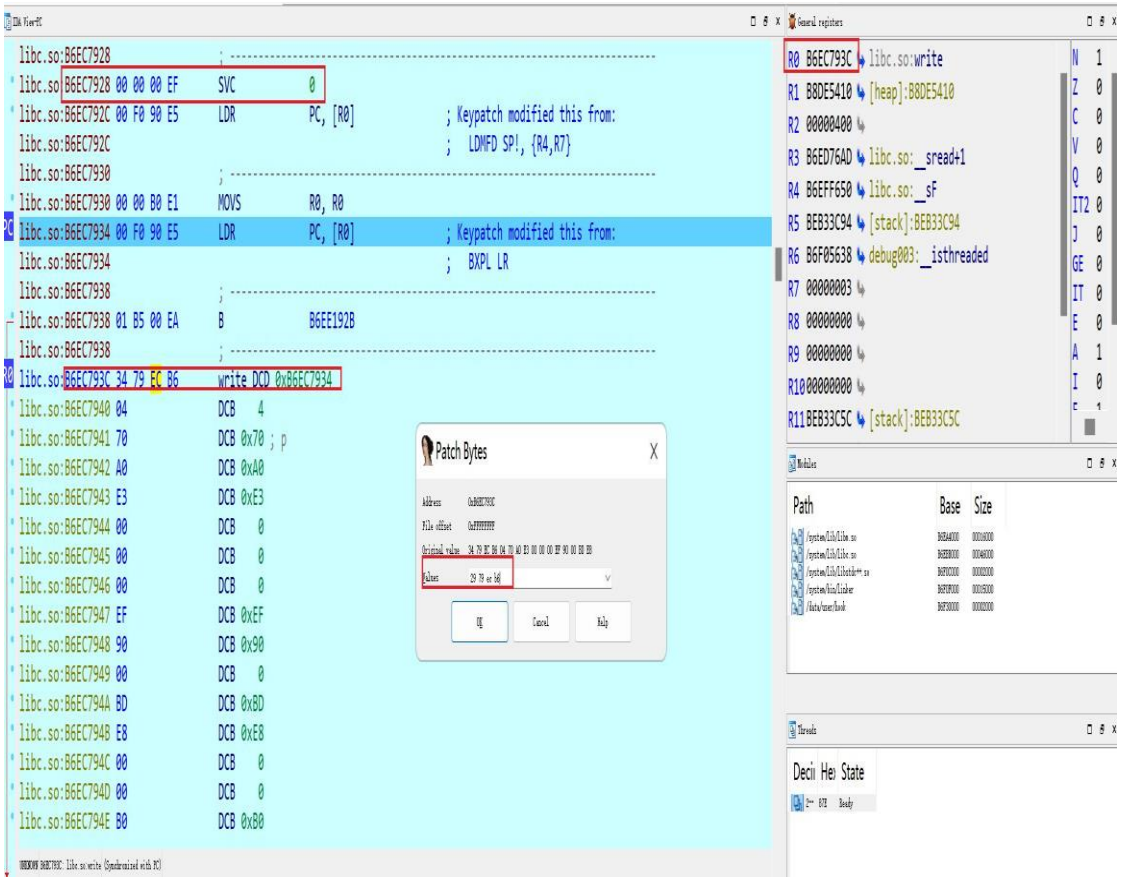
- MOV PC,REG（一般不用）：MOV 指令写 PC 会给到相应地址，但不会发生模式切换；
- BX REG：根据目标寄存器值的地址最低位是 0 还是 1，确定目标模式 T 标志位；
- LDR PC,[R0]直接从内存加载地址到 pc：把 r0 寄存器指向地址里的内存给 PC，分为两种情况：

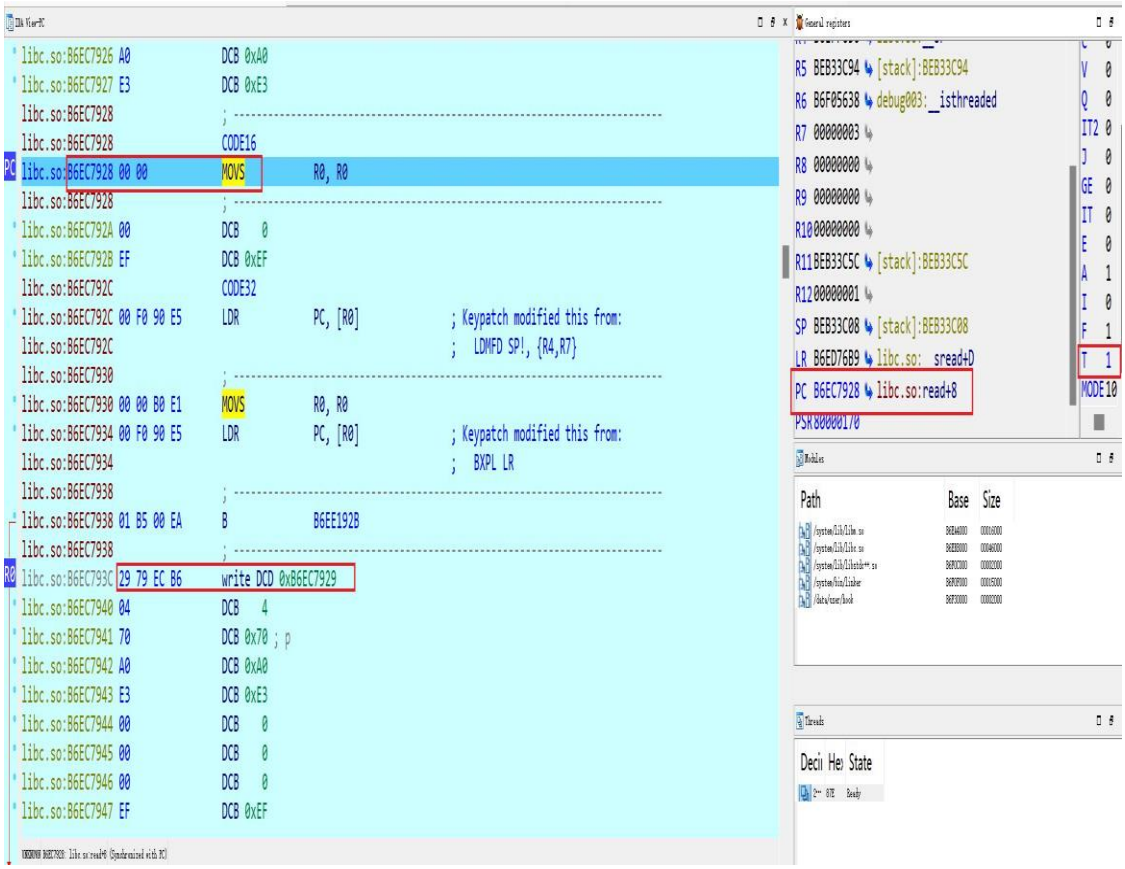
（1） 不发生模式切换：


libc.so:B6EC7928 ; ---
libc.so:B6EC7928 00 00 00 EF SVC 0
libc.so:B6EC792C 00 F0 90 E5 LDR PC, [R0] ; Keypatch modified this from:
libc.so:B6EC792C ; LDMFD SP!, {R4,R7}
libc.so:B6EC7930 ; ---
libc.so:B6EC7930 00 00 B0 E1 MOVS R0, R0
libc.so:B6EC7934 1E FF 2F 51 BXPL LR
libc.so:B6EC7938 01 B5 00 EA B B6EE192B
libc.so:B6EC7938 ; ---
libc.so:B6EC793C B4 79 EC B6 write DCD 0xB6EC7934
libc.so:B6EC7940 04 DCB 4
libc.so:B6EC7941 70 DCB 0x70 ; p
libc.so:B6EC7942 A0 DCB 0xA0
libc.so:B6EC7943 E3 DCB 0xE3
libc.so:B6EC7944 00 DCB 0
libc.so:B6EC7945 00 DCB 0
libc.so:B6EC7946 00 DCB 0
libc.so:B6EC7947 EF DCB 0xEF
libc.so:B6EC7948 90 DCB 0x90
libc.so:B6EC7949 00 DCB 0
libc.so:B6EC794A BD DCB 0xBD
libc.so:B6EC794B E8 DCB 0xE8
libc.so:B6EC794C 00 DCB 0
libc.so:B6EC794D 00 DCB 0
libc.so:B6EC794E B0 DCB 0xB0
libc.so:B6EC794F E1 DCB 0xE1
libc.so:B6EC7950 1E DCB 0x1E
Patch Bytes
R0 B6EC793C libc.so:write
R1 B8DE5410 [heap]:B8DE5410
R2 00000400
R3 B6ED76AD libc.so:__sread+1
R4 B6EFF650 libc.so:_sF
R5 BEB33C94 [stack]:BEB33C94
R6 B6F05638 debug003:_isthreaded
R7 00000003
R8 00000000
R9 00000000
R10 00000000
R11 BEB33C5C [stack]:BEB33C5C
Path Base Size


libc.so:B6EC7928 ; ---
libc.so:B6EC7928 00 00 00 EF SVC 0
libc.so:B6EC792C 00 F0 90 E5 LDR PC, [R0] ; Keypatch modified this from:
libc.so:B6EC792C ; LDMFD SP!, {R4,R7}
libc.so:B6EC7930 ; ---
libc.so:B6EC7930 00 00 B0 E1 MOVS R0, R0
libc.so:B6EC7934 1E FF 2F 51 BXPL LR
libc.so:B6EC7938 01 B5 00 EA B B6EE192B
libc.so:B6EC7938 ; ---
libc.so:B6EC793C B4 79 EC B6 write DCD 0xB6EC7934
libc.so:B6EC7940 04 DCB 4
libc.so:B6EC7941 70 DCB 0x70 ; p
libc.so:B6EC7942 A0 DCB 0xA0
libc.so:B6EC7943 E3 DCB 0xE3
libc.so:B6EC7944 00 DCB 0
libc.so:B6EC7945 00 DCB 0
libc.so:B6EC7946 00 DCB 0
libc.so:B6EC7947 EF DCB 0xEF
libc.so:B6EC7948 90 DCB 0x90
libc.so:B6EC7949 00 DCB 0
libc.so:B6EC794A BD DCB 0xBD
libc.so:B6EC794B E8 DCB 0xE8
libc.so:B6EC794C 00 DCB 0
libc.so:B6EC794D 00 DCB 0
libc.so:B6EC794E B0 DCB 0xB0
libc.so:B6EC794F E1 DCB 0xE1
libc.so:B6EC7950 1E DCB 0x1E
BXPL LR
B B6EE192B
; ---
write DCD 0xB6EC7934
DCB 4
DCB 0x70 ; p
DCB 0xA0
DCB 0xE3
DCB 0
DCB 0
R0 B6EC793C libc.so:write
R1 B8DE5410 [heap]:B8DE5410
R2 00000400
R3 B6ED76AD libc.so:__sread+1
R4 B6EFF650 libc.so:_sF
R5 BEB33C94 [stack]:BEB33C94
R6 B6F05638 debug003:_isthreaded
R7 00000003
R8 00000000
R9 00000000
R10 00000000
R11 BEB33C5C [stack]:BEB33C5C
Path Base Size

（2） 发生模式切换：R0 指向地址的内容更改为目标地址，并且其值加 1。（T 标志位变为 1，转换到 THUMB 模式）



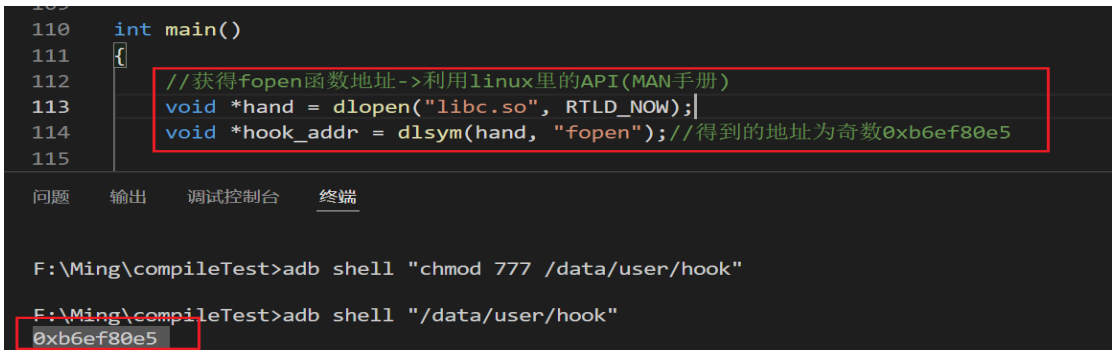


3 Inline Hook 实现

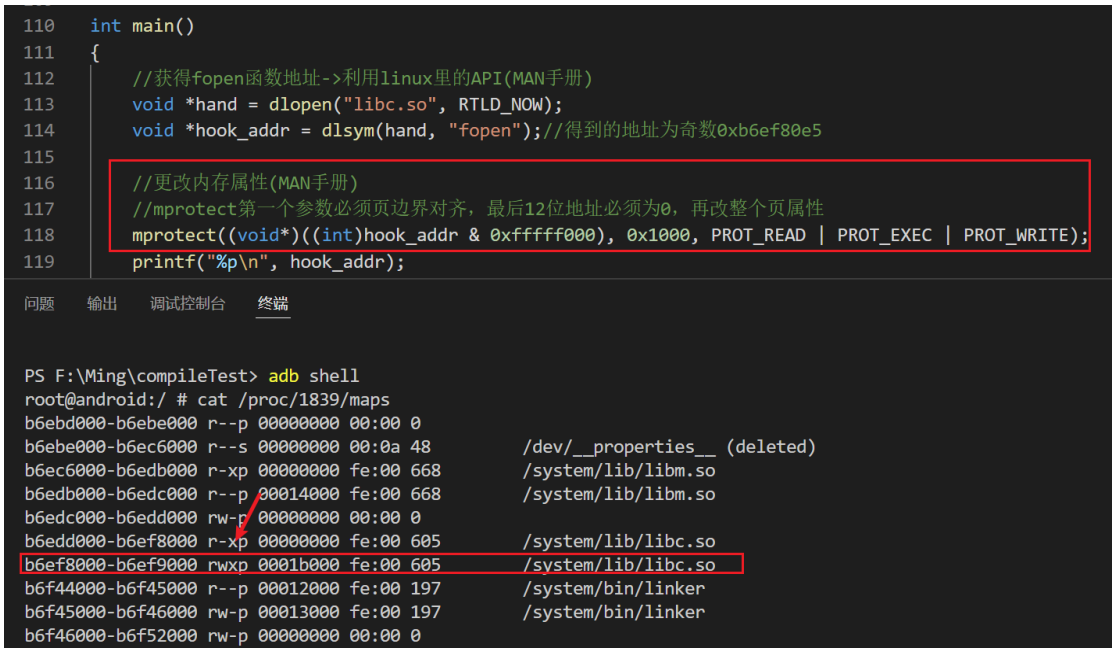
HOOK 实验目标为：对自身进程的 libc 库中的 fopen 函数进行 hook。其中为验证 hook 执行是否成功，在写好 hook 代码后，调用 fopen 函数，验证其正常执行流程无误。

同时在 hook 路径下，进入 fopen 函数传入相应的参数，并在该路径上测试传入的参数是否可以成功导出。

3.1 通过打开 libc.so 句柄获得 fopen 函数地址



3.2 更改内存属性



3.3 从 fopen 地址跳转到裸函数

```
121     //从fopen地址跳到裸函数
122     //真实地址: 要减1 -> 0001B0E4;  ldr读pc地址(hook_addr-1)->正好是下一条指令地址
123     *(uint32_t *)((uint32_t)hook_addr - 1) = 0xf000f8df;  //LDR PC,[PC] -> DF F8 00 F0
124
125     //LDR PC,[PC](THUMB) thumb模式下读PC要+4字节;  [pc] -> +4地址的内存（为nakefun地址）给pc;
126     //+4个字节为裸函数地址,即跳转到目标地址
127     *(uint32_t *)((uint32_t)hook_addr - 1 + 4) = (uint32_t)nakeFun;
```

```
libc.so:B6EFF0E4                    ; Attributes: thunk
libc.so:B6EFF0E4
libc.so:B6EFF0E4                    fopen
libc.so:B6EFF0E4 DF F8 00 F0        LDR.W           PC, =nakeFun
libc.so:B6EFF0E4                    ; End of function fopen
libc.so:B6EFF0E4
libc.so:B6EFF0E4                    ; ---------------------------------------------------------------------------
libc.so:B6EFF0E8 C0 96 F5 B6        off_B6EFF0E8 DCD nakeFun        ; DATA XREF: fopen↑r
libc.so:B6EFF0EC FF                 DCB 0xFF
libc.so:B6EFF0ED F7                 DCB 0xF7
libc.so:B6EFF0EE BA                 DCB 0xBA
libc.so:B6EFF0EF FF                 DCB 0xFF
libc.so:B6EFF0F0 05                 DCB    5
libc.so:B6EFF0F1 46                 DCB 0x46 ; F
```

```
.text:0001B0E4           ; =============== S U B R O U T I N E =======================================
.text:0001B0E4
.text:0001B0E4
.text:0001B0E4                           EXPORT fopen
.text:0001B0E4           fopen                                   ; CODE XREF: pututline+20↓p
.text:0001B0E4                                                   ; pututline+80↓p ...
.text:0001B0E4
.text:0001B0E4           var_14          = -0x14
.text:0001B0E4
.text:0001B0E4           ; __unwind {
.text:0001B0E4 DF F8 00 F0               LDR.W           PC, loc_1B0E8 ; Keypatch modified this from:
.text:0001B0E4                                                   ;   PUSH {R0,R1,R4-R6,LR}
.text:0001B0E4                                                   ;   MOV R6, R0
.text:0001B0E8           ; ---------------------------------------------------------------------------
.text:0001B0E8
.text:0001B0E8           loc_1B0E8                               ; DATA XREF: fopen↑r
.text:0001B0E8 08 46                     MOV             R0, R1
.text:0001B0EA 01 A9                     ADD             R1, SP, #0x18+var_14
.text:0001B0EC FF F7 BA FF               BL              __sflags
.text:0001B0F0 05 46                     MOV             R5, R0
.text:0001B0F2 04 46                     MOV             R4, R0
.text:0001B0F4 00 28                     CMP             R0, #0
.text:0001B0F6 2A D0                     BEQ             loc_1B14E
.text:0001B0F8 FF F7 26 FF               BL              __sfp
```

3.4 进行 inline hook

```
/*
        从fopen地址跳到裸函数->变成ARM模式
        裸函数可能是ARM模式也可能是Thumb模式->处理4类情况(理论上)ARM-ARM ARM-THUMB THUMB-ARM THUMB-THUMB
        实际考虑两类ARM-ARM THUMB-ARM ARM模式下
*/
void __attribute__((naked)) nakeFun(){
    //裸函数跳回去返回地址，用堆栈保存寄存器，寄存器状态保持不变，返回地址写到PC
    //将需要的push进栈（汇编代码需要的）
    asm("STMFD sp!, {R0,R1,R4-R6,LR}");// PUSH      {R0,R1,R4-R6,LR}

    //自己需要的push压栈
    asm("STMFD sp!, {R0-R12,LR,PC}");
    //保存通用寄存器CPSR
    asm("mrs r0, cpsr");
    asm("STMFD sp!, {R0}");
    ////=================================================================
    //获取fopen传入的两个参数
    asm("ldr r4, [sp, #4]");
    asm("str r4, %0":"=m"(arg0));
    asm("ldr r4, [sp, #8]");
    asm("str r4, %0":"=m"(arg1));
    ////-----------------------------------------------------------------
```

```

    asm("ldr r0, [sp, #4]");                    //pop  R0->R0
    asm("str r0, [sp, #0x1c]");                 //MOV  R6, R0
    asm("ldr r0, [sp, #8]");                    //pop  R1->R0
    asm("str r0, [sp, #4]");                    //MOV  R0, R1
    asm("mov r0, sp");                          //add  R1, SP, #4
    asm("add r0, r0, #0x44");
    asm("str r0, [sp, #8]");

    //ret_addr返回地址给R0
    asm("ldr r0, %0"::"m"(ret_addr));           //行列汇编语法格式==伪指令
    //sp与pc有3C的偏移，R0写到pc
    asm("str r0, [sp, #0x3c]");

    ////=================================================================
    asm("LDMFD sp!, {R0}");
    asm("msr cpsr, r0");
    //自己需要的栈平掉pop
    asm("LDMFD sp!, {R0-R12,LR,PC}");

}
```

```
hook:B6F596C0                                           ; DATA XREF: fopen1o ...
hook:B6F596C4 03 30 8F E0       ADD             R3, PC, R3          ; _GLOBAL_OFFSET_TABLE_
hook:B6F596C8 73 40 2D E9       STMFD           SP!, {R0,R1,R4-R6,LR}
hook:B6F596CC FF DF 2D E9       STMFD           SP!, {R0-R12,LR,PC}
hook:B6F596D0 00 00 0F E1       MRS             R0, CPSR
hook:B6F596D4 01 00 2D E9       STMFD           SP!, {R0}
hook:B6F596D8 04 40 9D E5       LDR             R4, [SP,#0x58+var_54]
hook:B6F596DC 50 20 9F E5       LDR             R2, =0xFFFFFFD0
hook:B6F596E0 02 20 93 E7       LDR             R2, [R3,R2]
hook:B6F596E4 00 40 82 E5       STR             R4, [R2]
hook:B6F596E8 08 40 9D E5       LDR             R4, [SP,#0x58+var_50]
hook:B6F596EC 44 20 9F E5       LDR             R2, =0xFFFFFFD4
hook:B6F596F0 02 20 93 E7       LDR             R2, [R3,R2]
hook:B6F596F4 00 40 82 E5       STR             R4, [R2]
hook:B6F596F8 04 00 9D E5       LDR             R0, [SP,#0x58+var_54]
hook:B6F596FC 1C 00 8D E5       STR             R0, [SP,#0x58+var_3C]
hook:B6F59700 08 00 9D E5       LDR             R0, [SP,#0x58+var_50]
hook:B6F59704 04 00 8D E5       STR             R0, [SP,#0x58+var_54]
hook:B6F59708 0D 00 A0 E1       MOV             R0, SP
hook:B6F5970C 44 00 80 E2       ADD             R0, R0, #0x44
hook:B6F59710 08 00 8D E5       STR             R0, [SP,#0x58+var_50]
hook:B6F59714 20 20 9F E5       LDR             R2, =0xFFFFFFD8
hook:B6F59718 02 30 93 E7       LDR             R3, [R3,R2]
hook:B6F5971C 00 00 93 E5       LDR             R0, [R3]
hook:B6F59720 3C 00 8D E5       STR             R0, [SP,#0x58+var_1C]
hook:B6F59724 01 00 BD E8       LDMFD           SP!, {R0}
hook:B6F59728 00 F0 29 E1       MSR             CPSR_cf, R0
hook:B6F5972C FF DF BD E8       LDMFD           SP!, {R0-R12,LR,PC}
hook:B6F5972C                   ; End of function nakeFun
```

hook目标地址代码块

```
hook:B6F59704 04 00 8D E5       STR             R0, [SP,#0x58+var_54]
hook:B6F59708 0D 00 A0 E1       MOV             R0, SP
hook:B6F5970C 44 00 80 E2       ADD             R0, R0, #0x44
hook:B6F59710 08 00 8D E5       STR             R0, [SP,#0x58+var_50]
hook:B6F59714 20 20 9F E5       LDR             R2, =0xFFFFFFD8
hook:B6F59718 02 30 93 E7       LDR             R3, [R3,R2]
hook:B6F5971C 00 00 93 E5       LDR             R0, [R3]
hook:B6F59720 3C 00 8D E5       STR             R0, [SP,#0x58+var_1C]
hook:B6F59724 01 00 BD E8       LDMFD           SP!, {R0}
hook:B6F59728 00 F0 29 E1       MSR             CPSR_cf, R0
hook:B6F5972C FF DF BD E8       LDMFD           SP!, {R0-R12,LR,PC}
hook:B6F5972C                   ; End of function nakeFun
hook:B6F5972C
hook:B6F5972C                   ; ---------------------------------------------------------------------------
hook:B6F59730 EC 38 00 00       off_B6F59730 DCD _GLOBAL_OFFSET_TABLE_ - 0xB6F596CC
hook:B6F59730                                                           ; DATA XREF: nakeFun↑r
hook:B6F59734 D0 FF FF FF       dword_B6F59734 DCD 0xFFFFFFD0           ; DATA XREF: nakeFun+1C↑r
hook:B6F59738 D4 FF FF FF       dword_B6F59738 DCD 0xFFFFFFD4           ; DATA XREF: nakeFun+2C↑r
hook:B6F5973C D8 FF FF FF       dword_B6F5973C DCD 0xFFFFFFD8           ; DATA XREF: nakeFun+54↑r
hook:B6F59740 10                main DCB 0x10
hook:B6F59741 48                DCB 0x48 ; H
hook:B6F59742 2D                DCB 0x2D ; -
hook:B6F59743 E9                DCB 0xE9
hook:B6F59744 08                DCB    8
hook:B6F59745 B0                DCB 0xB0
hook:B6F59746 8D                DCB 0x8D
hook:B6F59747 E2                DCB 0xE2
```

3.5 从裸函数跳回返回地址并验证 fopen 函数

```
23    //栈结构
24
25    /*
26    0         CPSR            <------SP
27    4         R0
28    8         R1
29    c         R2
30    10        R3
31    14        R4
32    18        R5
33    1c        R6
34    20        R7
35    24        R8
36    28        R9
37    2c        R10
38    30        R11
39    34        R12
40    38        LR
41    3c        PC
42    -------------------------
43    40        R0
44              R1
45              R4
46              R5
47              R6
48              LR
49    */
```

```
95      //ret_addr返回地址给R0
96      asm("ldr r0, %0"::"m"(ret_addr));          //行列汇编语法格式==伪指令
97      //sp与pc有3C的偏移，R0写到pc
98      asm("str r0, [sp, #0x3c]");
```

```
128     //返回地址->需要跳转回来的地址(ARM->THUMB 标志位改变 +1)->0001B0EC+1->0001B0ED
129     ret_addr = (uint32_t)hook_addr - 1 + 8 + 1;
```

```
.text:0001B0E4             ; =============== S U B R O U T I N E =======================================
.text:0001B0E4
.text:0001B0E4
.text:0001B0E4                             EXPORT fopen
.text:0001B0E4             fopen                                   ; CODE XREF: pututline+20↓p
.text:0001B0E4                                                     ; pututline+80↓p ...
.text:0001B0E4
.text:0001B0E4             var_14          = -0x14
.text:0001B0E4
.text:0001B0E4             ; __unwind {
.text:0001B0E4 73 B5                       PUSH            {R0,R1,R4-R6,LR}
.text:0001B0E6 06 46                       MOV             R6, R0
.text:0001B0E8 08 46                       MOV             R0, R1
.text:0001B0EA 01 A9                       ADD             R1, SP, #0x18+var_14
.text:0001B0EC FF F7 BA FF                 BL              __sflags
.text:0001B0F0 05 46                       MOV             R5, R0
.text:0001B0F2 04 46                       MOV             R4, R0
.text:0001B0F4 00 28                       CMP             R0, #0
.text:0001B0F6 2A D0                       BEQ             loc_1B14E
.text:0001B0F8 FF F7 26 FF                 BL              __sfp
```

```
libc.so:B6EFF0E0 21                DCB 0x21 ; !
libc.so:B6EFF0E1 BE                DCB 0xBE
libc.so:B6EFF0E2 70                DCB 0x70 ; p
libc.so:B6EFF0E3 47                DCB 0x47 ; G
libc.so:B6EFF0E4
libc.so:B6EFF0E4                   ; =============== S U B R O U T I N E =======================================
libc.so:B6EFF0E4
libc.so:B6EFF0E4                   ; Attributes: thunk
libc.so:B6EFF0E4
libc.so:B6EFF0E4                   fopen
libc.so:B6EFF0E4 DF F8 00 F0       LDR.W           PC, =nakeFun
libc.so:B6EFF0E4                   ; End of function fopen
libc.so:B6EFF0E4
libc.so:B6EFF0E4                   ; ---------------------------------------------------------------------------
libc.so:B6EFF0E8 C0 96 F5 B6       off_B6EFF0E8 DCD nakeFun            ; DATA XREF: fopen↑r
libc.so:B6EFF0EC                   ; ---------------------------------------------------------------------------
libc.so:B6EFF0EC FF F7 BA FF       BL              __sflags
libc.so:B6EFF0F0 05 46             MOV             R5, R0
libc.so:B6EFF0F2 04 46             MOV             R4, R0
libc.so:B6EFF0F4 00 28             CMP             R0, #0
libc.so:B6EFF0F6 2A D0             BEQ             loc_B6EFF14E
libc.so:B6EFF0F8 FF F7 26 FF       BL              __sfp
libc.so:B6EFF0FC 04 46             MOV             R4, R0
libc.so:B6EFF0FE 30 B3             CBZ             R0, loc_B6EFF14E
libc.so:B6EFF100 30 46             MOV             R0, R6
libc.so:B6EFF102 01 99             LDR             R1, [SP,#4]
libc.so:B6EFF104 4F F4 DB 72       MOV.W           R2, #0x1B6
UNKNOWN B6EFF0E4: fopen (Synchronized with PC)
```

```
130     while(1)
131     {
132         FILE *fp = fopen("/data/user/android_server", "rb");//android_server调试服务器-elf文件
133         uint32_t data;
134         fread(&data, 4, 1, fp);// 读前4个字节->elf签名
135         fclose(fp);
136         printf("data: %08x\n", data);
137
138         printf("arg0: %s\n", arg0);
139         printf("arg1: %s\n", arg1);
140         getchar();
141     }
142
```

```
F:\Ming\compileTest>adb push .\obj\local\armeabi-v7a\hook /data/user
.\obj\local\armeabi-v7a\hook: 1 file pushed, 0 skipped. 141.4 MB/s (45252 bytes in 0.000s)

F:\Ming\compileTest>adb shell "chmod 777 /data/user/hook"

F:\Ming\compileTest>adb shell "/data/user/hook"
0xb6f210e5
data: 464c457f
arg0: /data/user/android_server
arg1: rb
```

4 总结与思考

在本次 ARM 汇编的学习过程中，最初在环境搭建上花费了一些时间功夫。因为在环境搭建过程中需要考虑到模拟器 Emulator 的版本问题，故选择模拟器的 SDK 版本最好在 16 左右，太高版本可能无法运行 ARM 版 Android Emulator。

在编码过程中使用 7.0 版本的 IDA 作为调试工具，远程连接 IDA Server 对目标进程进行调试分析，同时在一些常用的 ARM 指令执行过程中，对遇到的寄存器以及栈变化进行记录学习，也为后面的 Inline Hook 实现打下基础。

在构建 Hook Stub 过程中，因为需要保存一些寄存器信息（包括 PC），使用 STMFD 和 LDMFD 实现进栈平栈，但在实际情况下使用 NDK21 编译会出现报错。经查询得知当下 ARM 已经抛弃了将 PC 放在 list 中使用，因此选择了降低 NDK21 版本至 NDK11 进行解决，最终成功编译执行 Inline Hook 代码。

虽然在整个学习过程中遇到了很多问题，但好在自己通过不断的努力和坚持解决了每一个编译报错。从对 ARM 汇编指令的熟悉到 Inline Hook 的实现，不仅是自己在编程能力和知识掌握上的技能提升，而且对董哲老师在课上所教学的嵌入式系统相关知识也有了更深一层的了解。

相信在以后的研究生课题研究中，这段时间的学习成果定会有所作为。

5 源码注释解读

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <dlfcn.h>
uint32_t ret_addr;
uint32_t arg0;
uint32_t arg1;
/*
        从fopen地址跳到裸函数->变成ARM模式
        裸函数可能是ARM模式也可能是Thumb模式->处理4类情况(理论上)ARM-ARM ARM-THUMB
THUMB-ARM THUMB-THUMB
        实际考虑两类ARM-ARM THUMB-ARM ARM模式下
*/
```

```
void __attribute__((naked)) nakeFun() {
    //裸函数跳回去返回地址，用堆栈保存寄存器，寄存器状态保持不变，返回地址写到PC
    //将需要的push进栈（汇编代码需要的）
    asm("STMFD sp!, {R0,R1,R4-R6,LR}");// PUSH       {R0,R1,R4-R6,LR}
    //自己需要的push压栈
    asm("STMFD sp!, {R0-R12,LR,PC}");
    //保存通用寄存器CPSR
    asm("mrs r0, cpsr");
    asm("STMFD sp!, {R0}");
    ////=====================================================================
    //获取fopen传入的两个参数
    asm("ldr r4, [sp, #4]");
    asm("str r4, %0":"=m"(arg0));
    asm("ldr r4, [sp, #8]");
    asm("str r4, %0":"=m"(arg1));
    ////---------------------------------------------------------------------
    asm("ldr r0, [sp, #4]");                    //pop  R0->R0
    asm("str r0, [sp, #0x1c]");                 //MOV  R6, R0
    asm("ldr r0, [sp, #8]");                    //pop  R1->R0
    asm("str r0, [sp, #4]");                    //MOV  R0, R1
    asm("mov r0, sp");                          //add  R1, SP, #4
    asm("add r0, r0, #0x44");
    asm("str r0, [sp, #8]");
    //ret_addr返回地址给R0
    asm("ldr r0, %0"::"m"(ret_addr));           //行列汇编语法格式==伪指令
    //sp与pc有3C的偏移，R0写到pc
    asm("str r0, [sp, #0x3c]");
    ////=====================================================================
    asm("LDMFD sp!, {R0}");
    asm("msr cpsr, r0");
    //自己需要的栈平掉pop
    asm("LDMFD sp!, {R0-R12,LR,PC}");
}
```

```
int main()
{
    //获得fopen函数地址->利用linux里的API(MAN手册)
    void *hand = dlopen("libc.so", RTLD_NOW);
    void *hook_addr = dlsym(hand, "fopen");//得到的地址为奇数0xb6ef80e5
    //更改内存属性(MAN手册)
    //mprotect第一个参数必须页边界对齐，最后12位地址必须为0，再改整个页属性
    mprotect((void*)((int)hook_addr & 0xfffff000), 0x1000, PROT_READ | PROT_EXEC |
PROT_WRITE);
    printf("%p\n", hook_addr);
    //从fopen地址跳到裸函数
    //真实地址：要减1 -> 0001B0E4;  ldr读pc地址(hook_addr-1)->正好是下一条指令地址
    *(uint32_t *)((uint32_t)hook_addr - 1) = 0xf000f8df;  //LDR PC,[PC] -> DF F8 00 F0
    //LDR PC,[PC](THUMB) thumb模式下读PC要+4字节;  [pc] -> +4地址的内存（为nakefun地址）
给pc;
    //＋4个字节为裸函数地址,即跳转到目标地址
    *(uint32_t *)((uint32_t)hook_addr - 1 + 4) = (uint32_t)nakeFun;
    //返回地址->需要跳转回来的地址(ARM->THUMB 标志位改变 +1)->0001B0EC+1->0001B0ED
    ret_addr = (uint32_t)hook_addr - 1 + 8 + 1;
    while (1)
    {
        FILE *fp = fopen("/data/user/android_server", "rb");//android_server调试服务器
-elf文件
        uint32_t data;
        fread(&data, 4, 1, fp);// 读前4个字节->elf签名
        fclose(fp);
        printf("data: %08x\n", data);
        printf("arg0: %s\n", arg0);
        printf("arg1: %s\n", arg1);
        getchar();
    }
    return 0;
}
```