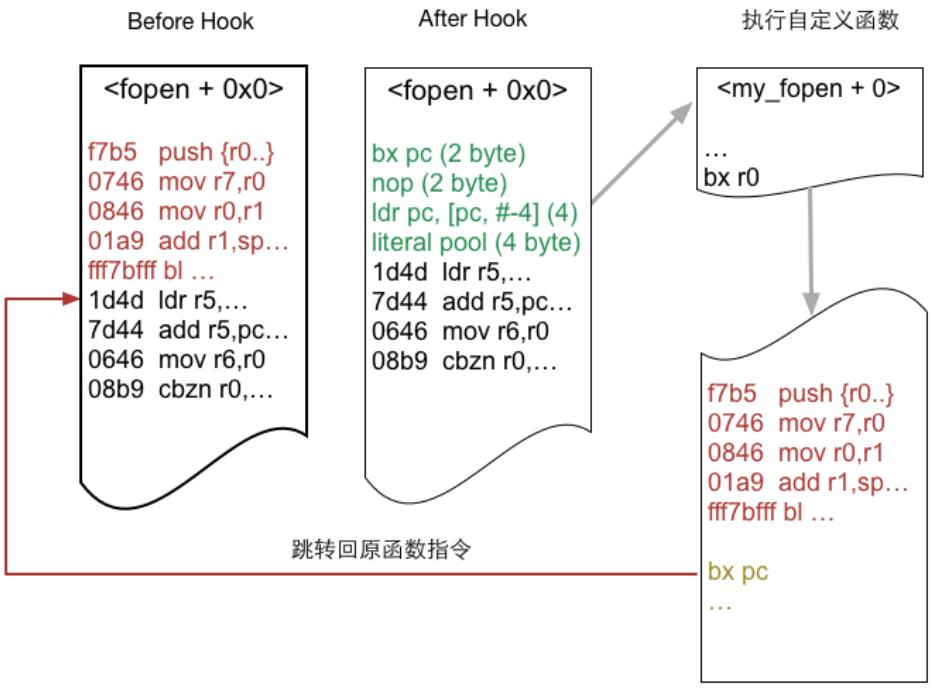
inline hook修改方案

1 inline hook基本原理



上图中描述了inline hook的基本原理。首先获取函数0x0开始的12或14个字节。以12字节为例，图中标红处，将前12个字节的指令存入buffer并对ldr, add, b, bl等相关指令进行重新定位；随后将fopen前12字节替换为标绿处的指令。当开发者调用fopen时，会先跳入自定义函数。自定义函数调用原始函数的时候会先执行buffer中的指令，最后通过bx跳回12个字节之后的指令。

但是对于某些函数本身不足12个字节，例如libdvm.so中的dvmIsClassObject或dvmIsClassVerified。

```
.text:00048856            ; _DWORD __fastcall dvmIsClassObject(const Object *)
.text:00048856                    EXPORT _Z16dvmIsClassObjectPK6Object
.text:00048856            _Z16dvmIsClassObjectPK6Object
.text:00048856 03 68              LDR             R3, [R0]
.text:00048858 18 6A              LDR             R0, [R3,#0x20]
.text:0004885A C0 F3 00 70        UBFX.W          R0, R0, #0x1C, #1
.text:0004885E 70 47              BX              LR
.text:0004885E            ; End of function dvmIsClassObject(Object const*)


.text:00048860            ; dvmIsTheClassClass(ClassObject const*)
.text:00048860                    EXPORT _Z18dvmIsTheClassClassPK11ClassObject
.text:00048860            _Z18dvmIsTheClassClassPK11ClassObject
.text:00048860 00 6A              LDR             R0, [R0,#0x20]
.text:00048862 C0 F3 00 70        UBFX.W          R0, R0, #0x1C, #1
.text:00048866 70 47              BX              LR
.text:00048866            ; End of function dvmIsTheClassClass(ClassObject const*)
```

2 inline hook改动一

因此需要将对前12或14个字节的改动改为前8或10个字节的改动。图中标绿色的指令总长度为12字节，实现的功能主要是跳转到自己定义的函数。但是完全可以用8个字节来实现。

修改为12个字节

```
thumb[0] = T$bx(A$pc);
thumb[1] = T$nop;

arm[0] = A$ldr_rd_$rn_im$(A$pc, A$pc, 4-8);
arm[1] = reinterpret_cast<uint32_t>(replace);
```

其对应的指令为：

```
bx pc
nop
ldr pc, [pc, #-4]
DCD
```

修改为8个字节

```
thumb[0] = 0xF85F;
thumb[1] = 0xF000;

arm[0] = reinterpret_cast<uint32_t>(replace);
```

其对应的指令为：

```
ldr pc, [pc, #-4]
DCD
```

3 switch...case...可能导致的问题

根据inline hook原理，将被hook的函数的前12或14字节保存在buffer中，并对其中需要重定位的指令进行重定位，之后在buffer中填充bx指令，再跳回到原函数的+12或者+14的位置处，继续执行原函数后面的指令。但是在类似以下的函数便出现了问题。

```
.text:00008AB4                                EXPORT AAssetManager_open
.text:00008AB4                AAssetManager_open
.text:00008AB4 10 B5                          PUSH            {R4,LR}
.text:00008AB6 03 2A                          CMP             R2, #3  ; switch 4 cases
.text:00008AB8 13 D8                          BHI             def_8ABA ; jumptable 00008ABA default case
.text:00008ABA DF E8 02 F0                    TBB.W           [PC,R2] ; switch jump
.text:00008ABA                                ; ---------------------------------------------------------------------------
.text:00008ABE 02                             jpt_8ABA        DCB 2                   ; jump table for switch statement
.text:00008ABF 08                                             DCB 8
.text:00008AC0 04                                             DCB 4
.text:00008AC1 06                                             DCB 6
.text:00008AC2                                ; ---------------------------------------------------------------------------
.text:00008AC2
.text:00008AC2                                loc_8AC2                                ; CODE XREF: AAssetManager_open+6↑j
.text:00008AC2 00 22                          MOVS            R2, #0  ; jumptable 00008ABA case 0
.text:00008AC4 04 E0                          B               loc_8AD0
```

```
AAsset* AAssetManager_open(AAssetManager* amgr, const
                           char* filename, int mode) {
    Asset::AccessMode amMode;
    switch (mode) {
    case AASSET_MODE_UNKNOWN:
        amMode = Asset::ACCESS_UNKNOWN;
        break;
    case AASSET_MODE_RANDOM:
        amMode = Asset::ACCESS_RANDOM;
        break;
    case AASSET_MODE_STREAMING:
        amMode = Asset::ACCESS_STREAMING;
        break;
    case AASSET_MODE_BUFFER:
        amMode = Asset::ACCESS_BUFFER;
        break;
    default:
        return NULL;
    }
    ……
}
```

函数是在0x8AB4开始的，如果挖掉前12个字节，很明显bx跳回来的时候则指向了0x8AC0处，此时无法解析指令，便导致了程序崩溃。

3.1 switch原理

3.1.1 switch构造

```
branchtable
DCB ((dest0 – branchtable)/2)
DCB ((dest1 – branchtable)/2)
DCB ((dest2 – branchtable)/2)
DCB ((dest3 – branchtable)/2)
dest0
... ; r0 = 0时执行
dest1
... ; r0 = 1时执行
dest2
... ; r0 = 2时执行
dest3
... ; r0 = 3时执行
```

在AAssetManager_open函数中，CMP R2, #3指令中，寄存器R2存储了传递的数值。

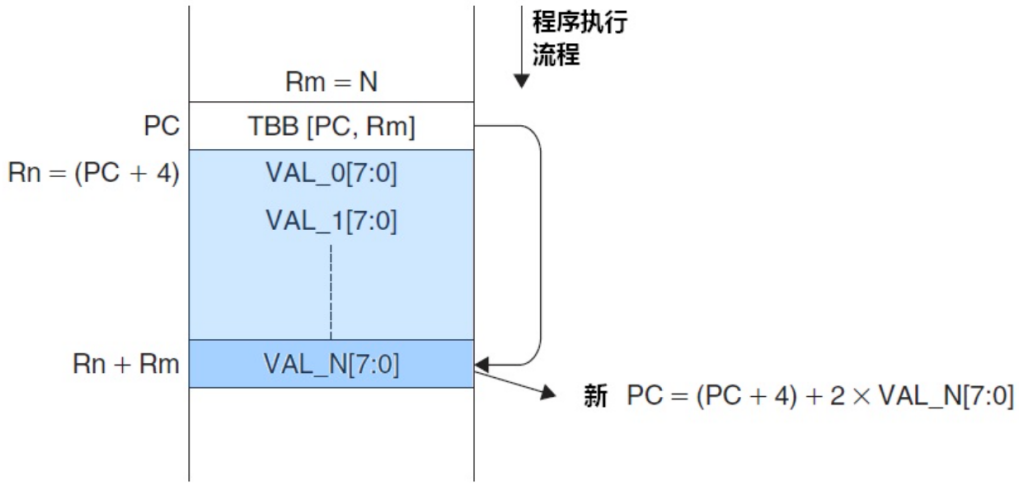
对于TBB.W [pc, r0]， 执行此指令时，PC的值正好等于branchtable。然后我们需要根据R2寄存器中的值来获取目标跳转地址。

3.1.2 目标跳转地址获取方式

那么，目标跳转地址的获取方式是什么呢?

从构造中可以得出R2为0的时候，跳转的目标地址为0x8AC2（2*2 + 0x8ABE）。

由于对该函数进行inline hook后，跳转到的地址为brancetable内部，因此需要手动根据传入的值，通过brancetable中的数值，进行计算进而得到要执行的下一条指令。下图中也表明了执行原理。其实要执行的下一条指令并不是TBB后面的值(PC+4)，而是根据branchtable计算出来的新PC。



3.2 解决方式

3.2.1 buffer构造

先查看存储原始指令的buffer中内容。

| original instructions |

|bx pc|

|nop|

|ldr pc, [pc, #-4]|; 标红处

|relocated address|； 标红处

|relocated addresses of original instructions|

只需要重新修改标红处即可。使得可以顺利执行下一条指令。

3.2.2 重定位方式

类似的函数有两种，第一种是前12个字节中不包含branchtable中的内容（libc.so中的sysconf），第二种是前12个字节中包含branchtable（libandroid.so中的AAssetManager_open等）。

```
.text:00013CEC 7F B5                    PUSH            {R0-R6,LR}
.text:00013CEE 64 28                    CMP             R0, #0x64 ; switch 101 cases
.text:00013CF0 00 F2 9C 80              BHI.W           def_13CF4 ; jumptable 00013CF4 default case
.text:00013CF4 DF E8 00 F0              TBB.W           [PC,R0] ; switch jump
.text:00013CF4                  ; ---------------------------------------------------------------------------
.text:00013CF8 33               jpt_13CF4       DCB 0x33                ; jump table for switch statement
.text:00013CF9 A1                               DCB 0xA1
.text:00013CFA 5E                               DCB 0x5E
.text:00013CFB A1                               DCB 0xA1
.text:00013CFC 36                               DCB 0x36
.text:00013CFD 39                               DCB 0x39
```

```
.text:00008AB4                          EXPORT AAssetManager_open
.text:00008AB4                  AAssetManager_open
.text:00008AB4 10 B5                    PUSH            {R4,LR}
.text:00008AB6 03 2A                    CMP             R2, #3  ; switch 4 cases
.text:00008AB8 13 D8                    BHI             def_8ABA ; jumptable 00008ABA default case
.text:00008ABA DF E8 02 F0              TBB.W           [PC,R2] ; switch jump
.text:00008ABA                  ; ---------------------------------------------------------------------------
.text:00008ABE 02               jpt_8ABA        DCB 2                   ; jump table for switch statement
.text:00008ABF 08                               DCB 8
.text:00008AC0 04                               DCB 4
.text:00008AC1 06                               DCB 6
.text:00008AC2                  ; ---------------------------------------------------------------------------
```

在sysconf或者AAssetManager_open中各有一个寄存器R0或R2。这两个寄存器保存着branchtable中某项的偏移。故计算跳转目标地址必须使用该寄存器的值进行计算。

计算方式：

```
if((T$pcrel$byte$tbb(area + 0x4) || T$pcrel$byte$tbb(area + 0x3))
         && T$pcrel$byte$cmp$imm(*(area+1))) {

        /* wipe away the TBB instruction. replace it with "bx pc; nop" */
        buffer[start-1] = T$bx(A$pc);
        buffer[start-2] = T$nop;

        uint32_t *transfer = reinterpret_cast<uint32_t *>(buffer + start);
        int reg = *((unsigned char*)area + 3) & 0x07;
        int A$add$r9$reg = 0xe0899000 | reg;

        /* read (base + [offset]*2) */
        transfer[0] = 0xe92d0c00;   /* push {r10,r11} */
        transfer[1] = 0xe59f9018;   /* ldr r9, [pc,#0x18] */
        transfer[2] = 0xe59fa014;   /* ldr r10, [pc,#0x14] */
        transfer[3] = A$add$r9$reg; /* add r9, reg */
        transfer[4] = 0xe5d9b000;   /* ldrb r11, [r9] */
        transfer[5] = 0xe08aa08b;   /* add r10, r11, LSL #1 */
        transfer[6] = 0xe28a9001;   /* add r9, r10, #1 */
        transfer[7] = 0xe8bd0c00;   /* pop {r10, r11}  */
        transfer[8] = 0xe12fff19;   /* bx r9 */

        if(T$pcrel$byte$tbb(area + 0x4)) {
                transfer[9] = reinterpret_cast<uint32_t>(area+used/sizeof(uint16_t));
        } else if(T$pcrel$byte$tbb(area + 0x3)) {
                transfer[9] = reinterpret_cast<uint32_t>(area+used/sizeof(uint16_t)-1);
        }

} else {
        buffer[start++] = T$bx(A$pc);
        buffer[start++] = T$nop;

        uint32_t *transfer = reinterpret_cast<uint32_t *>(buffer + start);
        transfer[0] = A$ldr_rd_$rn_im$(A$pc, A$pc, 4-8);
        transfer[1] = reinterpret_cast<uint32_t>(area+used/sizeof(uint16_t)) + 1;
}
```

算法：

1. 首先判断原函数第二条以及第四条指令是否为“带立即数的CMP”和“TBB”。
2. 通过CMP指令获取寄存器名称
3. 计算跳转地址，*(branchtable基地址 + 寄存器中数值)*2 + branchtabletable基地址
4. 最后是获取branchtable的基地址，对sysconf和AAssetManager_open分别进行计算。

4 inline hook改动二

由于将前12或14个字节的改动改为8或10个字节，故对于sysconf函数来说无需进行修改。但是类似于AAssetManager_open此类函数，需要获取前10个字节，因此仍需对TBB指令进行重定位。

```
if(T$pcrel$byte$tbb(area + 0x3) && T$pcrel$byte$cmp$imm(*(area+1))) {

        /* wipe away the TBB instruction. replace it with "bx pc; nop" */
        buffer[start-1] = T$bx(A$pc);
        buffer[start-2] = T$nop;

        uint32_t *transfer = reinterpret_cast<uint32_t *>(buffer + start);
        int reg = *((unsigned char*)area + 3) & 0x07;
        int A$add$r9$reg = 0xe0899000 | reg;

        /* read (base + [offset]*2) */
        transfer[0] = 0xe92d0c00;   /* push {r10,r11} */
        transfer[1] = 0xe59f9018;   /* ldr r9, [pc,#0x18] */
        transfer[2] = 0xe59fa014;   /* ldr r10, [pc,#0x14] */
        transfer[3] = A$add$r9$reg; /* add r9, reg */
        transfer[4] = 0xe5d9b000;   /* ldrb r11, [r9] */
        transfer[5] = 0xe08aa08b;   /* add r10, r11, LSL #1 */
        transfer[6] = 0xe28a9001;   /* add r9, r10, #1 */
        transfer[7] = 0xe8bd0c00;   /* pop {r10, r11}  */
        transfer[8] = 0xe12fff19;   /* bx r9 */

        if(T$pcrel$byte$tbb(area + 0x3)) {
                transfer[9] = reinterpret_cast<uint32_t>(area+used/sizeof(uint16_t));
        }

 } else {
        buffer[start++] = T$bx(A$pc);
        buffer[start++] = T$nop;

        uint32_t *transfer = reinterpret_cast<uint32_t *>(buffer + start);
        transfer[0] = A$ldr_rd_$rn_im$(A$pc, A$pc, 4-8);
        transfer[1] = reinterpret_cast<uint32_t>(area+used/sizeof(uint16_t)) + 1;
}
```