软件安全实验报告

姓名：郭子涵 学号：2312145 班级：信息安全、法学双学位班

目录：

1 实验名称

程序插桩及 Hook 实验

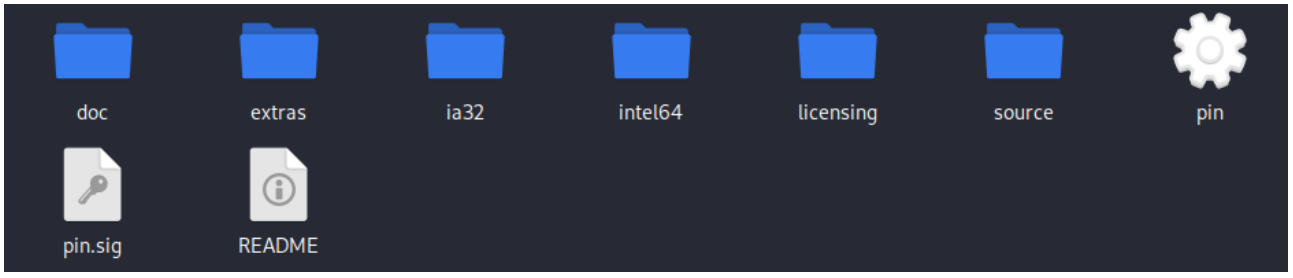
2 实验要求

复现实验一，基于 WindowsMyPinTool 或在 Kali 中复现 malloctrace 这个 PinTool，理解 Pin 插桩工具的核心步骤和相关 API，关注 malloc 和 free 函数的输入输出信息。

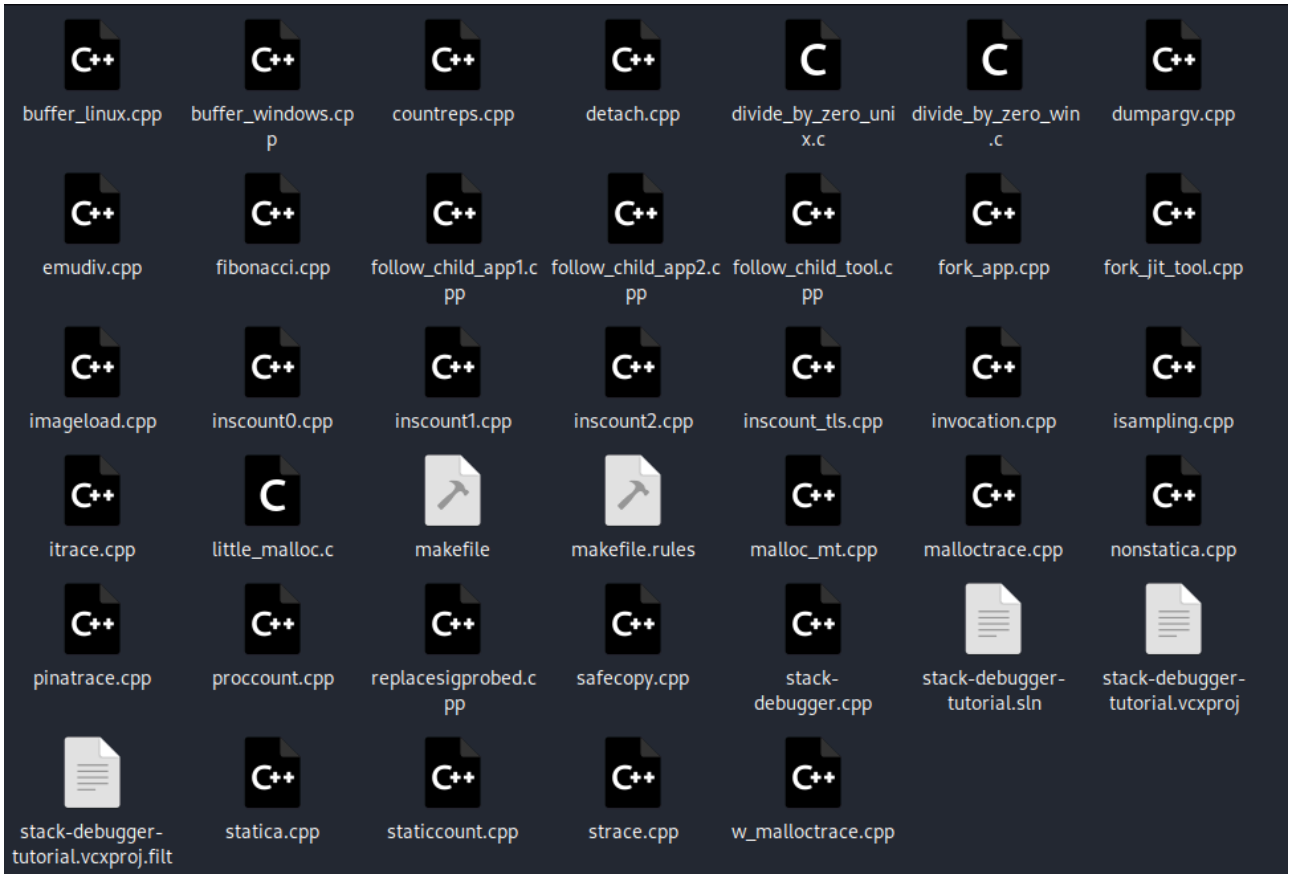
3 实验内容

3.1 Kali虚拟机中安装Pin

下载 Pin 的压缩包并解压:



进入 source/tools/ManualExamples 观察已有的 PinTool 程序

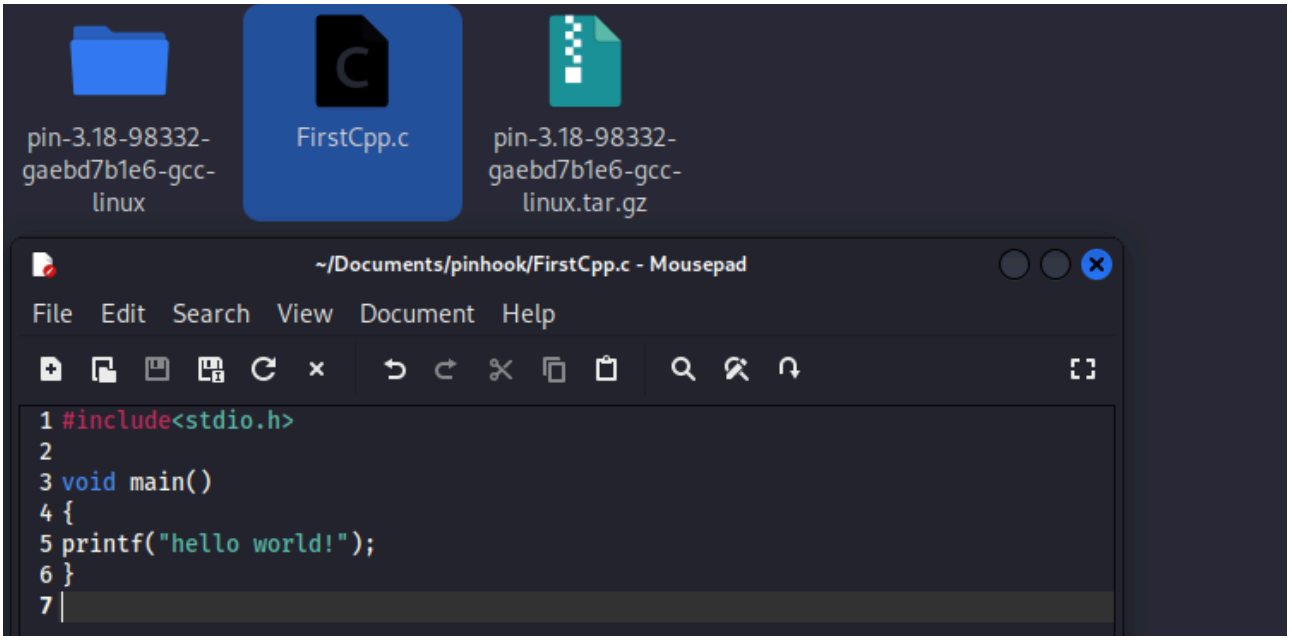


3.2 使用PinTool-inscount.0

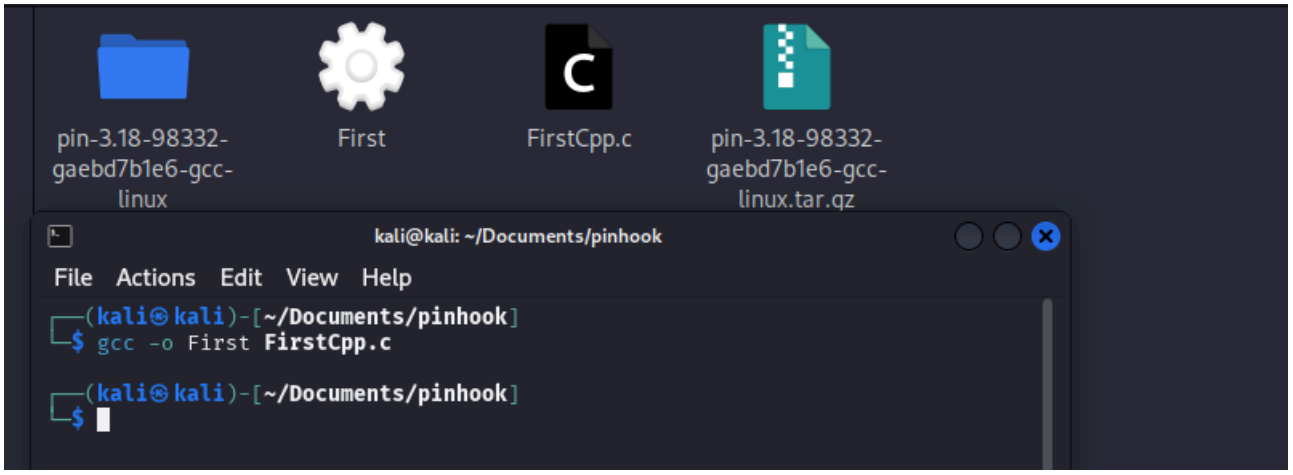
编译inscount.0:

```
e -isystem /home/kali/Documents/pin-3.18-98332-gaebd7b1e6-gcc-linux/extras/crt/include -isystem /home/kali/Documents/pi
n-3.18-98332-gaebd7b1e6-gcc-linux/extras/crt/include/arch-x86_64 -isystem /home/kali/Documents/pin-3.18-98332-gaebd7b1e
6-gcc-linux/extras/crt/include/kernel/uapi -isystem /home/kali/Documents/pin-3.18-98332-gaebd7b1e6-gcc-linux/extras/crt
/include/kernel/uapi/asm-x86 -I../../../extras/components/include -I../../../extras/xed-intel64/include/xed -I../../../
source/tools/Utils -I../../../source/tools/InstLib -O3 -fomit-frame-pointer -fno-strict-aliasing  -c -o obj-intel64/ins
count0.o inscount0.cpp
g++ -shared -Wl,--hash-style=sysv  ../../../intel64/runtime/pincrt/crtbeginS.o -Wl,-Bsymbolic -Wl,--version-script=../..
/../source/include/pin/pintool.ver -fabi-version=2   -o obj-intel64/inscount0.so obj-intel64/inscount0.o -L../../../int
el64/runtime/pincrt -L../../../intel64/lib -L../../../intel64/lib-ext -L../../../extras/xed-intel64/lib -lpin -lxed  ../
../../intel64/runtime/pincrt/crtendS.o -lpin3dwarf -ldl-dynamic -nostdlib -lstlport-dynamic -lm-dynamic -lc-dynamic -lu
nwind-dynamic
make -C  ../../../source/tools/Utils dir obj-intel64/cp-pin.exe
make[1]: Entering directory '/home/kali/Documents/pin-3.18-98332-gaebd7b1e6-gcc-linux/source/tools/Utils'
mkdir -p obj-intel64/
g++ -DTARGET_IA32E -DHOST_IA32E -DFUND_TC_TARGETCPU=FUND_CPU_INTEL64 -DFUND_TC_HOSTCPU=FUND_CPU_INTEL64 -DTARGET_LINUX
-DFUND_TC_TARGETOS=FUND_OS_LINUX -DFUND_TC_HOSTOS=FUND_OS_LINUX -I../../../source/tools/Utils -O3  -o obj-intel64/cp-pi
n.exe cp-pin.cpp -no-pie
make[1]: Leaving directory '/home/kali/Documents/pin-3.18-98332-gaebd7b1e6-gcc-linux/source/tools/Utils'
../../../pin   -t obj-intel64/inscount0.so --  ../../../source/tools/Utils/obj-intel64/cp-pin.exe makefile obj-intel64/i
nscount0.makefile.copy \
  > obj-intel64/inscount0.out 2>&1
cmp makefile obj-intel64/inscount0.makefile.copy
rm obj-intel64/inscount0.makefile.copy
```

编写 FirstCpp.c 程序:



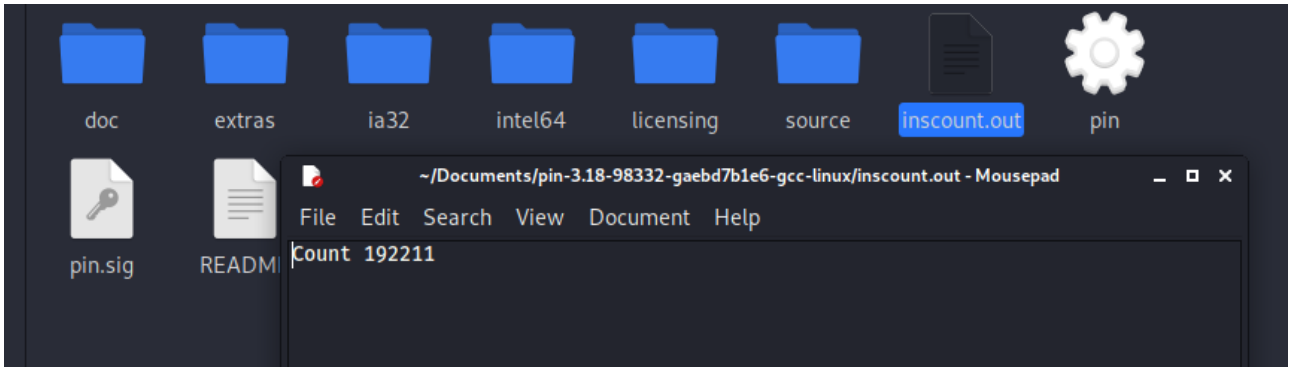
输入命令 gcc -o First FirstCpp.c 编译



对FirstCpp执行程序插桩命令为:

./pin -t ./source/tools/ManualExamples/obj-intel64/inscount0.so -- ../testCPP/First

成功执行后可以得到一个输出文件 **inscount.out**，文件内容如下，“**Count 192211**”即对指令数进行了插桩。

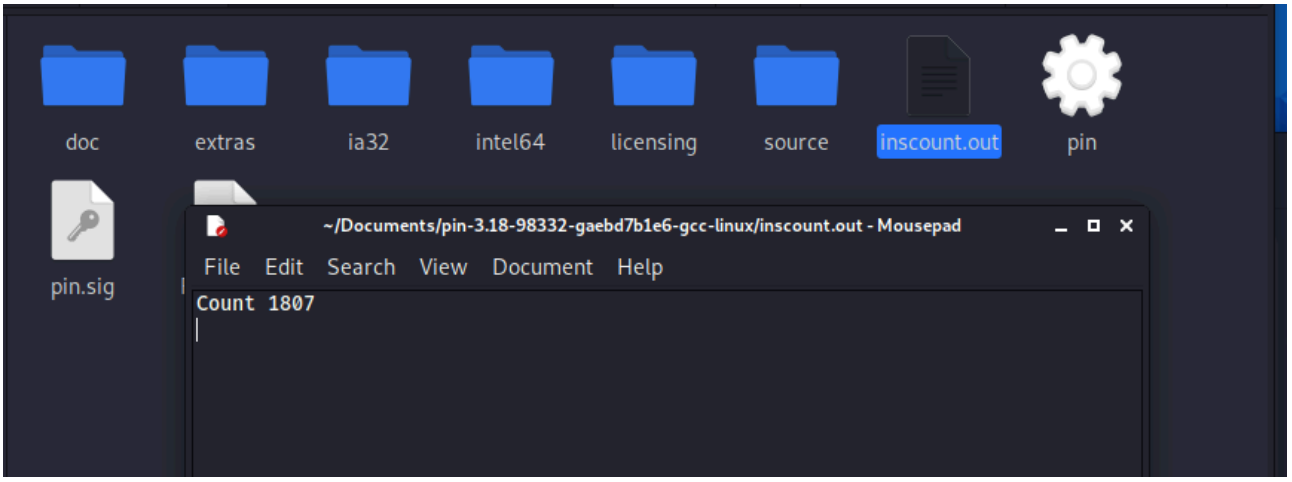


修改 **inscount.0** 插桩程序的代码，设置复杂的指令插桩条件，只有当满足命令是mov指令，是一条内存读指令，指令的第一个操作数为寄存器，第二个操作数是内存时才会计数。

```
//pin 在每次遇到新指令时调用此函数
VOID Instruction(INS ins, VOID *v)
{
```

```
    // 在每条指令之前插入对 docount 的调用，不传递任何参数
    //INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)docount, IARG_END);

    if (INS_Opcode(ins) == XED_ICLASS_MOV && //操作码对应mov指令
    INS_IsMemoryRead(ins) && //十一条内存读指令
    INS_OperandIsReg(ins,0) && //指令的第一个操作数为寄存器
    INS_OperandIsMemory(ins,1)) //指令的第二个操作数为内存
    {
        icount++; //计数
    }
}
```

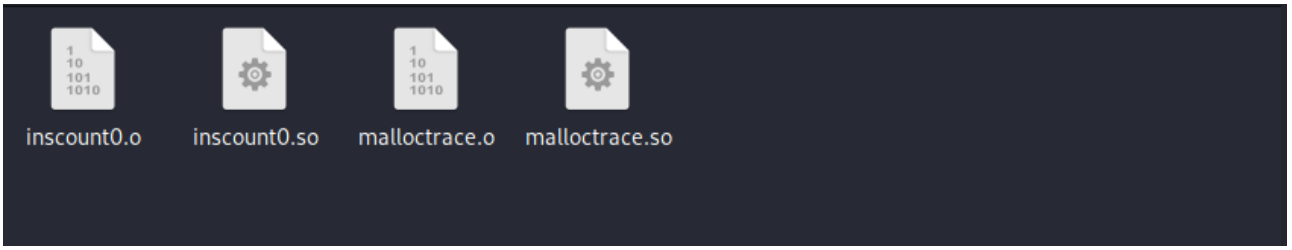
修改代码后队员程序插桩之后的结果为“Count 1807”可见指令数大大减少。



3.3 使用PinTool-mallotrace

首先编译 mallotrace.test，得到 mallotrace.so

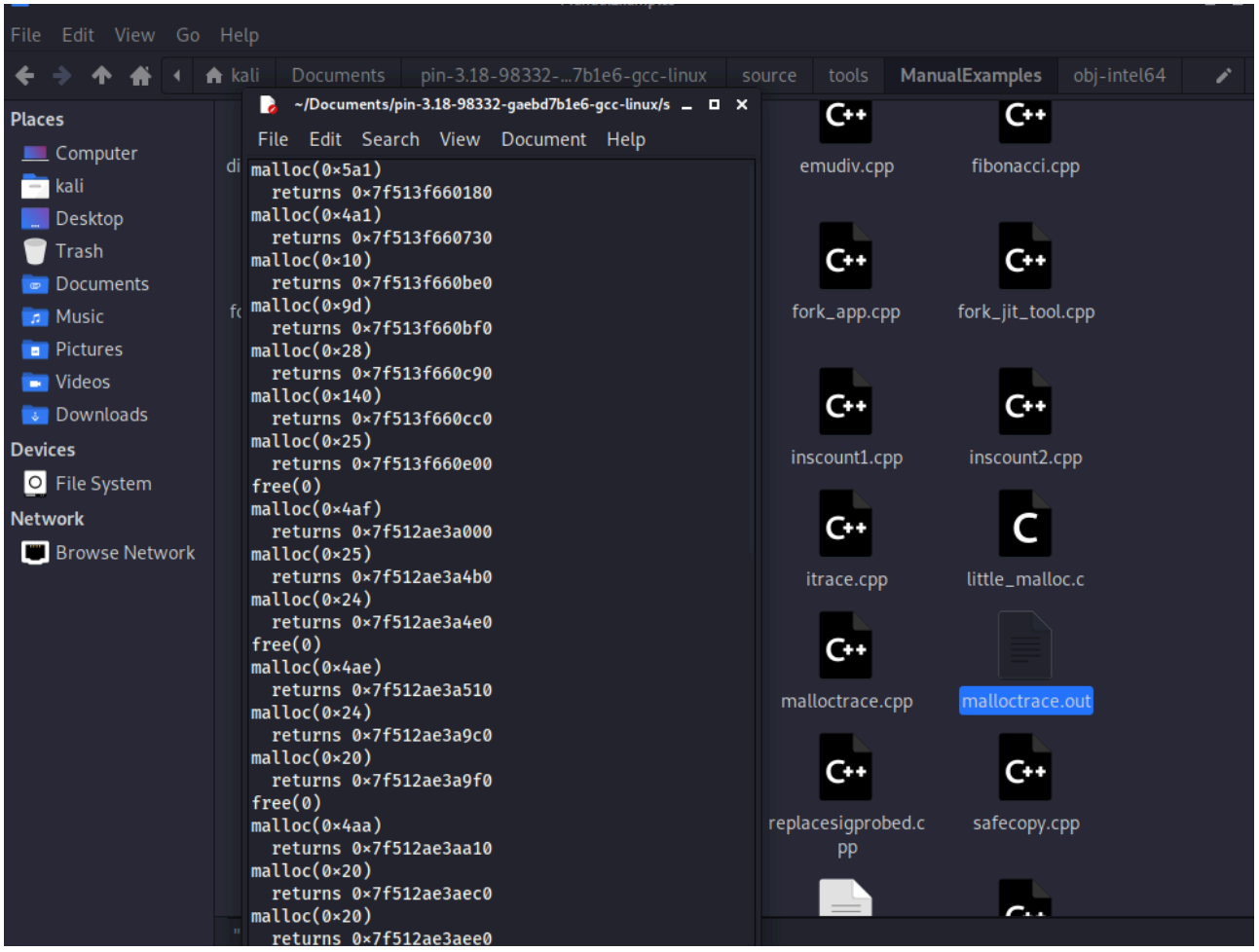
```
┌──(kali㉿kali)-[~/…/pin-3.18-98332-gaebd7b1e6-gcc-linux/source/tools/ManualExamples]
└─$ make malloctrace.test TARGET=intel64
g++ -Wall -Werror -Wno-unknown-pragmas -D__PIN__=1 -DPIN_CRT=1 -fno-stack-protector -fno-exceptions -funwind-tables -fa
synchronous-unwind-tables -fno-rtti -DTARGET_IA32E -DHOST_IA32E -fPIC -DTARGET_LINUX -fabi-version=2 -faligned-new -I..
/../../source/include/pin -I../../../source/include/pin/gen -isystem /home/kali/Documents/pin-3.18-98332-gaebd7b1e6-gcc
-linux/extras/stlport/include -isystem /home/kali/Documents/pin-3.18-98332-gaebd7b1e6-gcc-linux/extras/libstdc++/includ
e -isystem /home/kali/Documents/pin-3.18-98332-gaebd7b1e6-gcc-linux/extras/crt/include -isystem /home/kali/Documents/pi
n-3.18-98332-gaebd7b1e6-gcc-linux/extras/crt/include/arch-x86_64 -isystem /home/kali/Documents/pin-3.18-98332-gaebd7b1e
6-gcc-linux/extras/crt/include/kernel/uapi -isystem /home/kali/Documents/pin-3.18-98332-gaebd7b1e6-gcc-linux/extras/crt
/include/kernel/uapi/asm-x86 -I../../../extras/components/include -I../../../extras/xed-intel64/include/xed -I../../../
source/tools/Utils -I../../../source/tools/InstLib -O3 -fomit-frame-pointer -fno-strict-aliasing  -c -o obj-intel64/mal
loctrace.o malloctrace.cpp
g++ -shared -Wl,--hash-style=sysv  ../../../intel64/runtime/pincrt/crtbeginS.o -Wl,-Bsymbolic -Wl,--version-script=../..
/../source/include/pin/pintool.ver -fabi-version=2   -o obj-intel64/malloctrace.so obj-intel64/malloctrace.o -L../../..
/intel64/runtime/pincrt -L../../../intel64/lib -L../../../intel64/lib-ext -L../../../extras/xed-intel64/lib -lpin -lxed
 ../../../intel64/runtime/pincrt/crtendS.o -lpin3dwarf -ldl-dynamic -nostdlib -lstlport-dynamic -lm-dynamic -lc-dynamic
 -lunwind-dynamic
../../../pin   -t obj-intel64/malloctrace.so --  ../../../source/tools/Utils/obj-intel64/cp-pin.exe makefile obj-intel64
/malloctrace.makefile.copy \
  > obj-intel64/malloctrace.out 2>&1
cmp makefile obj-intel64/malloctrace.makefile.copy
rm obj-intel64/malloctrace.makefile.copy
rm obj-intel64/malloctrace.out
```



对原来的程序进行程序插桩

./pin -t ./source/tools/ManualExamples/obj-intel64/mallotrace.so -- ../testCPP/First

得到 mallotrace.out 的输出如下：



3.4 mallotrace代码分析

它的目标是 **跟踪应用程序中** `malloc()` **和** `free()` **的调用情况**，
并且把每次调用的参数和返回值记录到一个日志文件里。

3.4.1 引入头文件和命名空间

```
#include "pin.H"//- PIN工具开发必须包含这个头文件，里面定义了所有 PIN API。
#include <iostream>
#include <fstream>//写日志文件用
using std::hex;
using std::cerr;
using std::string;
using std::ios;
using std::endl;
```

3.4.2 定义宏：malloc 和 free 的名字

```
#if defined(TARGET_MAC)
#define MALLOC "_malloc"
#define FREE "_free"
#else
#define MALLOC "malloc"
#define FREE "free"
#endif
```

因为不同系统下，函数名字不一样，macOS下符号前面多了个下划线。这个宏保证了跨平台兼容，在 Linux 上就是 `"malloc"`、`"free"`。

3.4.3 全局变量：输出文件

```
std::ofstream TraceFile;//定义了一个文件输出流，用来写跟踪结果
```

3.4.4 定义命令行参数 Knob

```
KNOB<string> KnobOutputFile(KNOB_MODE_WRITEONCE, "pintool",
    "o", "malloctrace.out", "specify trace file name");
```

KNOB 是 PIN 里面用来处理命令行选项的机制。这里定义了一个选项 `-o`，可以指定输出文件名，比如下述代码：不指定的话，默认是 malloctrace.out。

```
pin -t obj-intel64/malloctrace.so -o mytrace.out -- myprogram
```

3.4.5 分析函数：打印参数和返回值

打印 malloc/free 参数

```
VOID Arg1Before(CHAR * name, ADDRINT size)
{
    TraceFile << name << "(" << size << ")" << endl;
}
```

这个函数在 `malloc()` 或 `free()` 被调用"之前"执行。它把调用的函数名和第一个参数（`malloc(size)` 里的size，或者 `free(ptr)` 里的ptr）打印出来。

打印 malloc 返回值

```
VOID MallocAfter(ADDRINT ret)
{
    TraceFile << "  returns " << ret << endl;
}
```

在 `malloc()` 调用"之后"执行。它记录 `malloc` 返回的内存地址。

3.4.6 Image 加载回调：找到 malloc 和 free

```
VOID Image(IMG img, VOID *v)
{
    RTN mallocRtn = RTN_FindByName(img, MALLOC);
    if (RTN_Valid(mallocRtn))
    {
        RTN_Open(mallocRtn);
        RTN_InsertCall(mallocRtn, IPOINT_BEFORE, (AFUNPTR)Arg1Before,
                       IARG_ADDRINT, MALLOC,
                       IARG_FUNCARG_ENTRYPOINT_VALUE, 0,
                       IARG_END);
        RTN_InsertCall(mallocRtn, IPOINT_AFTER, (AFUNPTR)MallocAfter,
                       IARG_FUNCRET_EXITPOINT_VALUE, IARG_END);
        RTN_Close(mallocRtn);
    }

    RTN freeRtn = RTN_FindByName(img, FREE);
    if (RTN_Valid(freeRtn))
    {
        RTN_Open(freeRtn);
        RTN_InsertCall(freeRtn, IPOINT_BEFORE, (AFUNPTR)Arg1Before,
                       IARG_ADDRINT, FREE,
                       IARG_FUNCARG_ENTRYPOINT_VALUE, 0,
                       IARG_END);
        RTN_Close(freeRtn);
    }
}
```

1. Image(IMG img, VOID *v) 是 PIN 在**每个映像模块加载**时调用的函数（比如程序本体、动态库）。
2. RTN_FindByName(img, MALLOC)：在这个模块里找函数 malloc。
3. RTN_Valid()：检查找没找到。
4. RTN_Open()/ RTN_Close()：在插桩前要open/close，表示修改这个Routine。
5. RTN_InsertCall() ：在函数执行前或后插入我们自己的代码。IPOINT_BEFORE`：函数执行前插入（打印输入参数）。IPOINT_AFTER：函数执行后插入（打印返回值）。
6. IARG_ADDRINT, MALLOC：传入函数名。
7. IARG_FUNCARG_ENTRYPOINT_VALUE, 0：传入第0个参数（即malloc的size）。
8. IARG_FUNCRET_EXITPOINT_VALUE：返回值（malloc的返回指针）。

3.4.7 程序结束时回调

```
VOID Fini(INT32 code, VOID *v)//当被测试程序结束时，这个函数会被调用，记得关掉文件。
{
    TraceFile.close();
}
```

3.4.8 打印帮助信息

```
INT32 Usage()//如果命令行参数不对，会调用这个 Usage() 打印帮助。
{
    cerr << "This tool produces a trace of calls to malloc." << endl;
    cerr << endl << KNOB_BASE::StringKnobSummary() << endl;
    return -1;
}
```

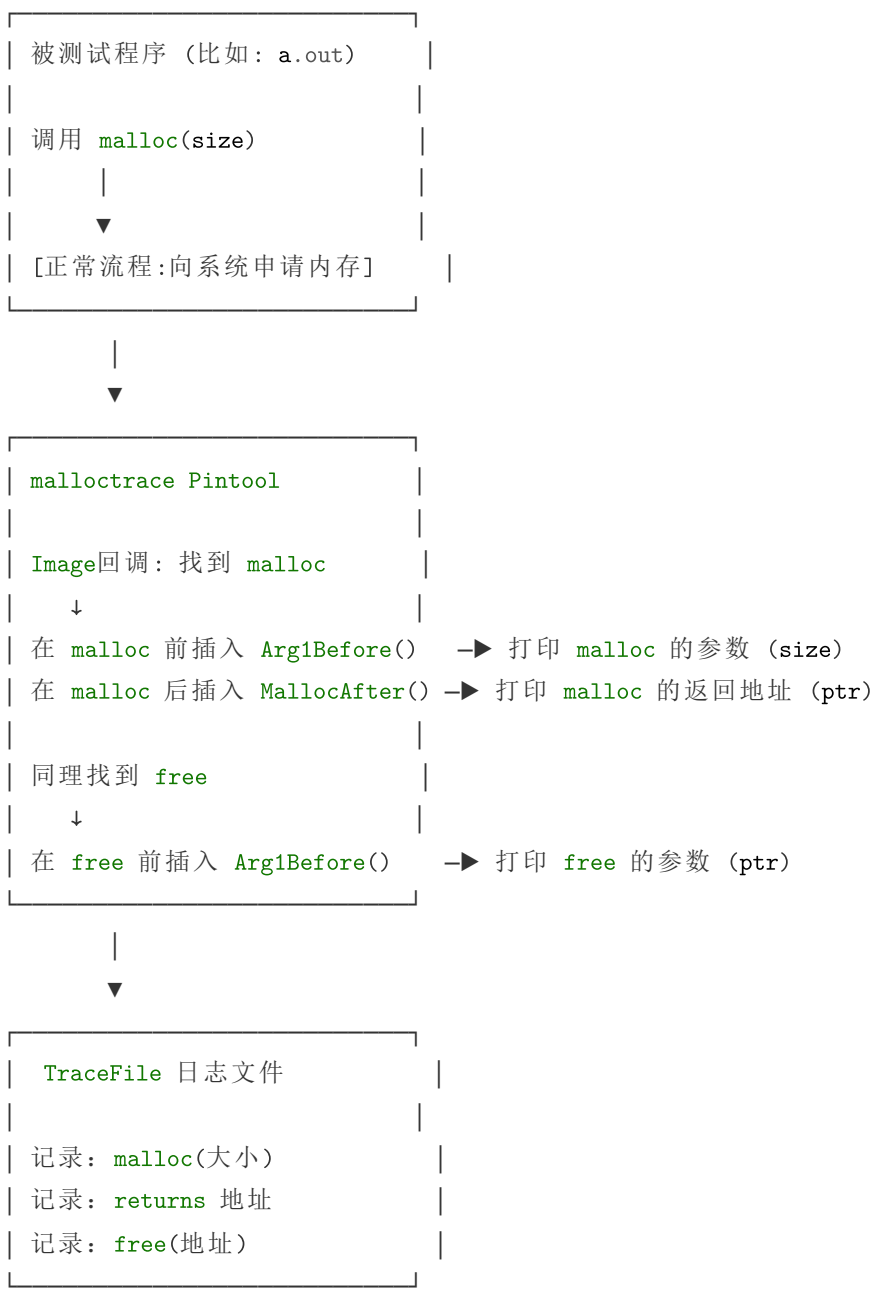
3.4.9 主函数 main

```
int main(int argc, char *argv[])
{
    // Initialize pin & symbol manager
    PIN_InitSymbols();
    if( PIN_Init(argc,argv) )
    {
        return Usage();
    }

    TraceFile.open(KnobOutputFile.Value().c_str());
    TraceFile << hex;
    TraceFile.setf(ios::showbase);

    // Register Image to be called to instrument functions.
    IMG_AddInstrumentFunction(Image, 0);
    PIN_AddFiniFunction(Fini, 0);

    // Never returns
    PIN_StartProgram();

    return 0;
}
```

1. PIN_InitSymbols()：初始化符号表，确保能找函数名。
2. PIN_Init()：初始化PIN环境（处理参数），如果失败打印Usage。
3. TraceFile.open()：打开输出文件。
4. IMG_AddInstrumentFunction(Image, 0)：告诉PIN，每加载一个模块就回调Image()，插桩。

5. PIN_AddFiniFunction(Fini, 0)：告诉PIN，程序结束时回调Fini()。
6. PIN_StartProgram()：开始真正执行被测程序，从这里不会返回（控制权交给了程序）。

3.5 mallotrace插桩流程简图



3.6 输出解析

这些输出表示程序在运行过程中，每次调用malloc（size）->mallocrate拦截并记录了：申请了多少字节和返回的内存地址；欸此调用free(ptr)->mallotrace拦截并记录了：要释放哪个地址（ptr）

```
malloc(0x5a1)
  returns 0x7f513f660180
malloc(0x4a1)
```

```
  returns 0x7f513f660730
malloc(0x10)
  returns 0x7f513f660be0
malloc(0x9d)
  returns 0x7f513f660bf0
malloc(0x28)
  returns 0x7f513f660c90
malloc(0x140)
  returns 0x7f513f660cc0
malloc(0x25)
  returns 0x7f513f660e00
free(0)//在C标准里free(NULL) 是合法的，不做任何操作。程序试图释放一个空指针（可能是异常处理、初始化时释放NULL，是正常现象）
malloc(0x4af)//地址慢慢线性增长，每次分配内存的时候返回稍微靠后一点的地址，符合堆区连续增长的特性。
  returns 0x7f512ae3a000
malloc(0x25)
  returns 0x7f512ae3a4b0
malloc(0x24)
  returns 0x7f512ae3a4e0
free(0)
malloc(0x4ae)
  returns 0x7f512ae3a510
malloc(0x24)
  returns 0x7f512ae3a9c0
malloc(0x20)
  returns 0x7f512ae3a9f0
free(0)
malloc(0x4aa)
  returns 0x7f512ae3aa10
malloc(0x20)
  returns 0x7f512ae3aec0
malloc(0x20)
  returns 0x7f512ae3aee0
free(0)
malloc(0x4aa)
  returns 0x7f512ae3af00
malloc(0x20)
  returns 0x7f512ae3b3b0
malloc(0x58)
  returns 0x7f512ae3b3d0
malloc(0x28)
  returns 0x7f512ae3b430
malloc(0x28)
  returns 0x7f512ae3b460
malloc(0x38)
  returns 0x7f512ae3b490
```

```
malloc(0x68)
  returns 0x7f512ae3b4d0
malloc(0xd8)
  returns 0x7f512ae3b540
malloc(0x48)
  returns 0x7f512ae3b620
malloc(0x5b8)
  returns 0x7f512ae3b670
malloc(0x180)
  returns 0x7f512ae3bc30
malloc(0x348)//程序在申请比较大的块的内存，可能在加载一些比较大的数据结构入缓冲区等
  returns 0x7f512a8e6000
malloc(0x1b0)
  returns 0x7f512a8e6350
malloc(0x90)
  returns 0x7f512a8e6500
malloc(0x420)
  returns 0x7f512a8e6590
malloc(0x1088)
  returns 0x7f512a8e69b0
malloc(0x120)
  returns 0x7f512a8e7a40
malloc(0x3e0)
  returns 0x7f512a8e7b60
malloc(0x7a0)//分配了 0x7a0（1952字节），地址 0x7f512a868000。释放了 地址 0x7f512a8e7b60
（之前的一个内存块）。然后又分配了 0xfe0（4064字节），拿到了一个稍微靠后的新地址。推测堆
内存正在动态管理，系统回收释放的空间或新分配一块足够大的内存。
  returns 0x7f512a868000
free(0x7f512a8e7b60)
malloc(0xfe0)
  returns 0x7f512a8687a0
free(0x7f512a868000)
malloc(0x11c00)
malloc(0x11c00)//两次连续申请 0x11c00。只有第二次有返回值记录原因可能是：可能是第一次
malloc实际上失败了（返回了NULL），但因为Arg1Before和MallocAfter分别在前后插入，第一次
malloc可能没有正常执行到返回，所以trace里没记录成功。
  returns 0xa842a0
malloc(0x208)
  returns 0xa95eb0
malloc(0x1d8)
  returns 0xa960c0
malloc(0x2000)
  returns 0xa962a0
malloc(0x200)
  returns 0xa982b0
malloc(0x1d8)
  returns 0xa984c0
```

```
malloc(0x2000)
  returns 0xa986a0
free(0xa962a0)//在最后，程序主动释放之前申请的一些内存
free(0xa960c0)
free(0xa986a0)
free(0xa984c0)
```

总结输出整体流程：

时间线	发生事件
开始阶段	程序申请了一些小块内存（几十到上千字节）
中间阶段	程序释放了NULL（free(0)），继续申请新的内存
中后期	申请了更大的内存（几百字节到上万字节），部分释放
最后阶段	释放了多个分配过的大块内存，程序在收尾

4 总结与心得

在本次实验中，围绕对内存动态管理过程的插桩与分析展开，学习使用并修改了inscount0.test插桩程序，以及mallotrace.test主要通过拦截malloc和free函数调用，记录每一次内存分配与释放的详细信息，最终生成了完整的内存操作轨迹（trace）。

通过本次实验，我对malloc和free这两个标准库函数的实际运行时行为有了更直观的认识。不再仅仅停留在它们的抽象调用层面，而是深入到每一次申请多少字节，返回什么样的地址，free又是释放哪个具体地址这样的底层细节。这种具体、量化的数据使我更加理解了堆内存的动态变化过程，比如内存是如何从低地址向高地址逐渐增长，如何在释放之后可能被回收或重新利用，又比如在程序进入不同阶段时，内存使用模式的变化——初始化阶段频繁的小块申请，大数据处理阶段突发的大块申请等等。这种现象也让我认识到，内存分配模式实际上反映了程序运行逻辑的结构性变化，可以作为程序动态分析和优化的重要线索。

此外，本次实验中我注意到free(0)的出现，并通过查阅资料确认了这是符合C标准行为的正常操作，加深了对C标准库行为细节的理解。这实际操作中还遇到了小量trace数据不完整的问题（例如某次malloc缺少返回值记录），也促使我反思了插桩程序在数据同步、缓冲区刷新等环节的严谨性，这对于今后做更复杂的插桩、动态分析乃至调试工具开发都有重要意义。

总体来说，这次实验不仅复现了几个小型的动态分析工具，还系统性训练了对程序执行流程、内存管理、系统调用拦截机制的理解和分析能力。“看见”malloc和free真实行为，理解内存变化的实际图景，让我对系统软件和程序运行机制有了更扎实、更深入的体会。