

《软件安全》实验报告

姓名：陆皓喆 学号：2211044 班级：信息安全

实验名称：

程序插桩及Hook实验

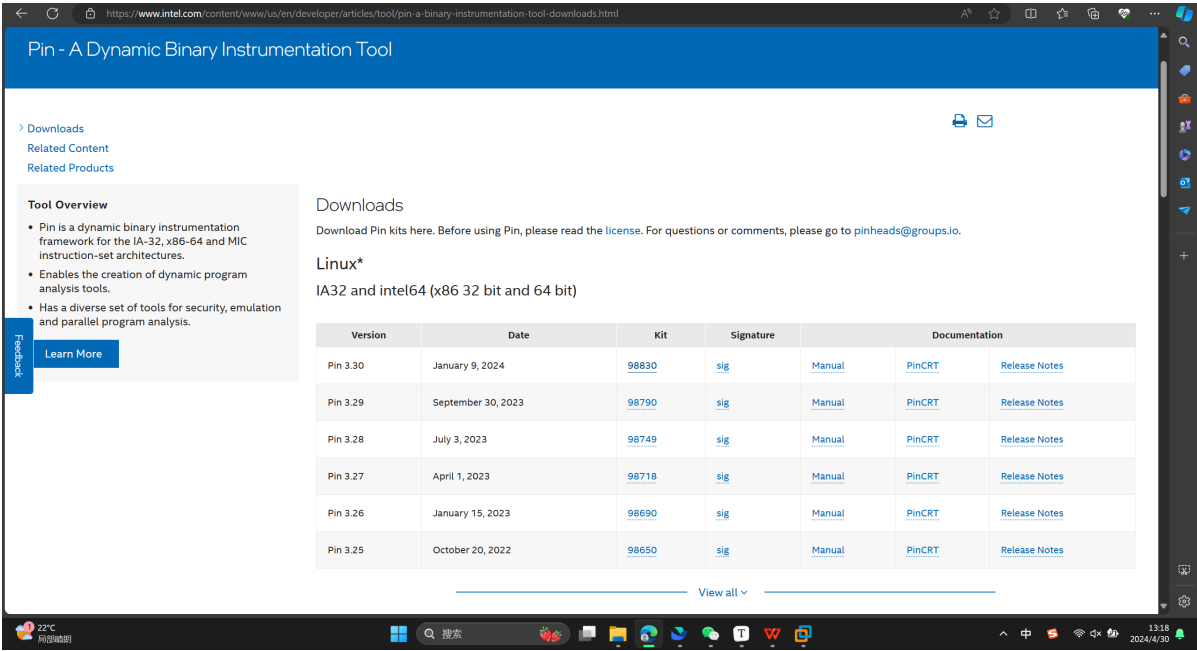
实验要求：

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。

实验过程：

1 在kali虚拟机中安装Pin

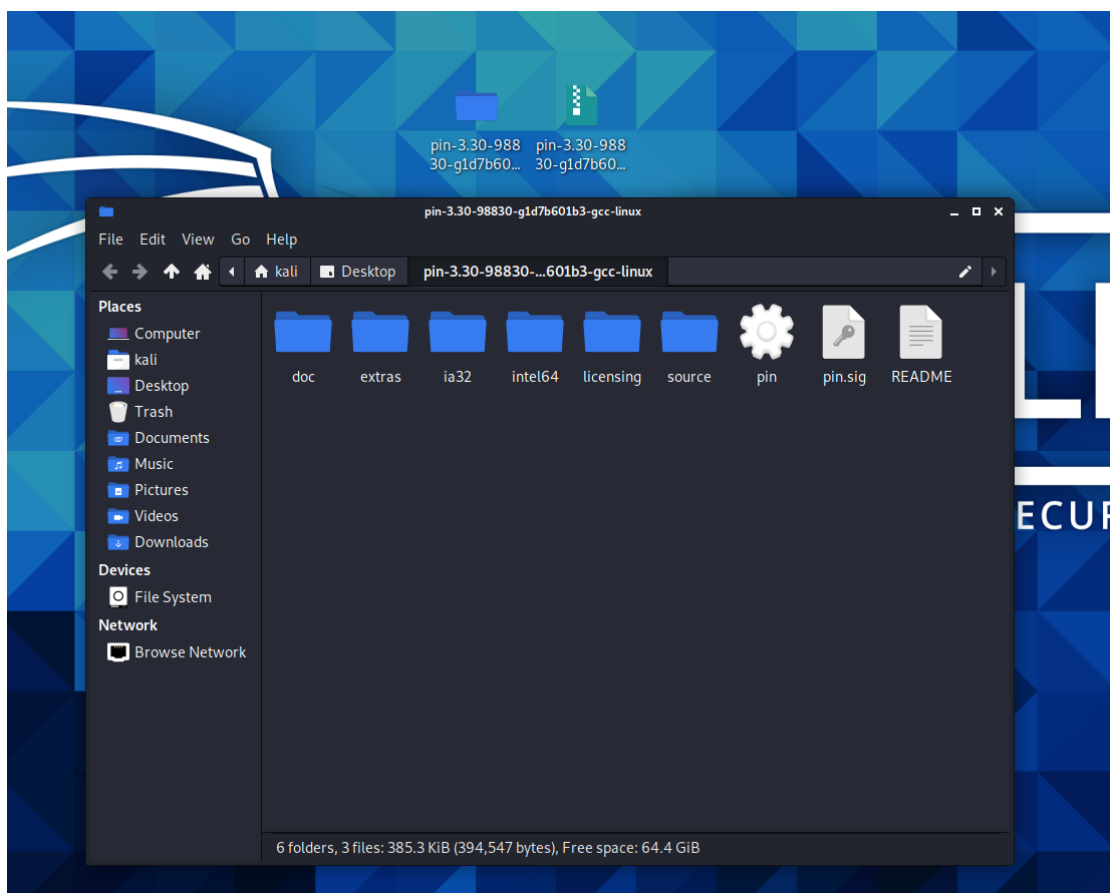
首先，我们需要安装Pin，但是因为我们在虚拟机上连不上网，所以我选择了在windows系统下先进行下载，然后压缩包拖到linux系统中。



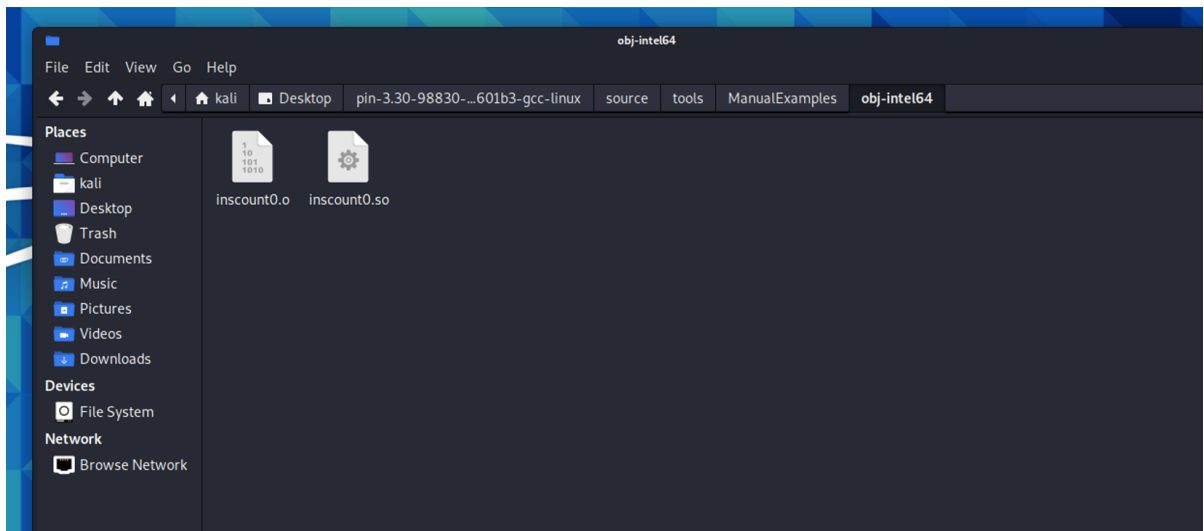
如图所示，我们选择最新版的进行安装即可，版本号为Pin3.30。然后我们将其拖入kali虚拟机，并完成解压工作。



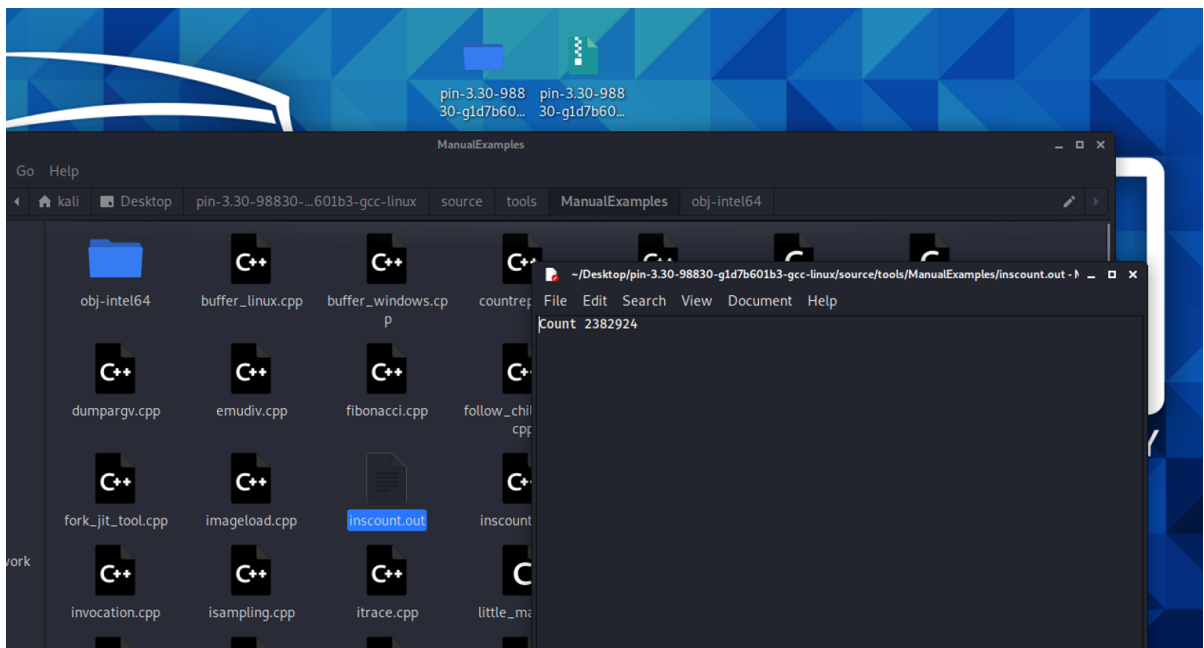
我们打开文件夹，发现里面有以下这些文件。



我们按照给出的路径，打开文件夹中的`source`中的`tools`的`ManualExamples`，就可以查看到很多已经编写好的`pintool`，如下所示：

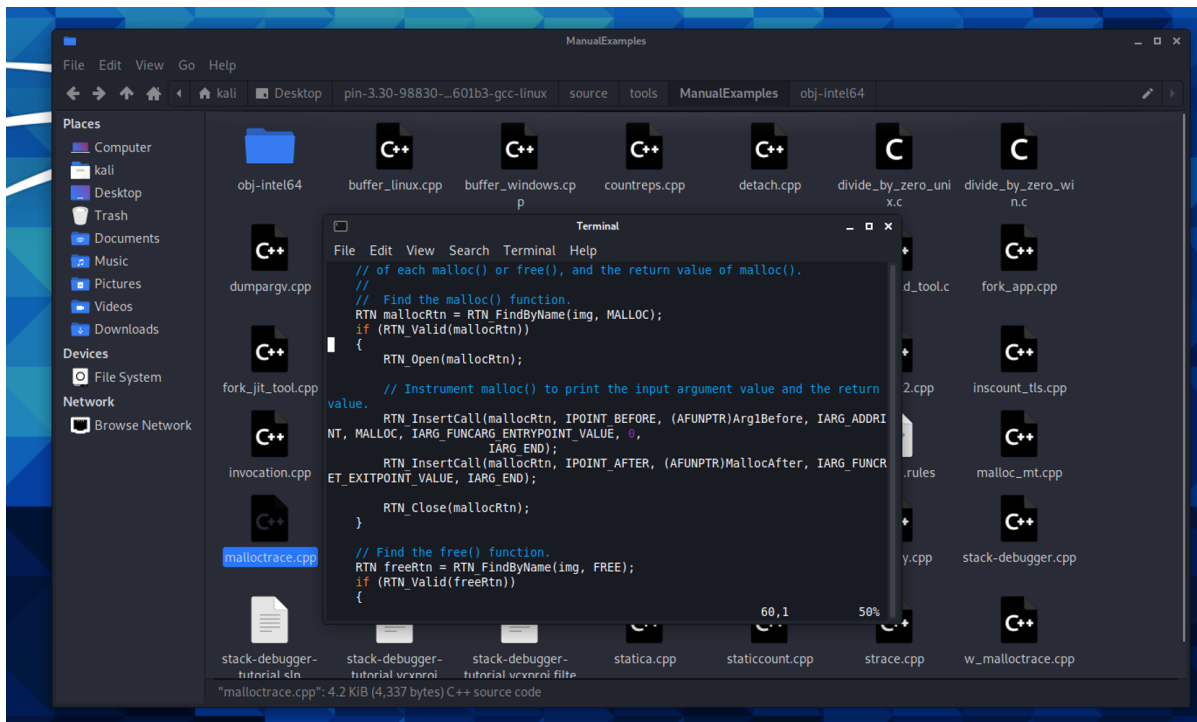


我们还可以查看编译运行后的 `out` 文件来查看编译所使用的指令数。我们打开文件 `inscount.out`，就可以看到输出的 `count` 数量了。

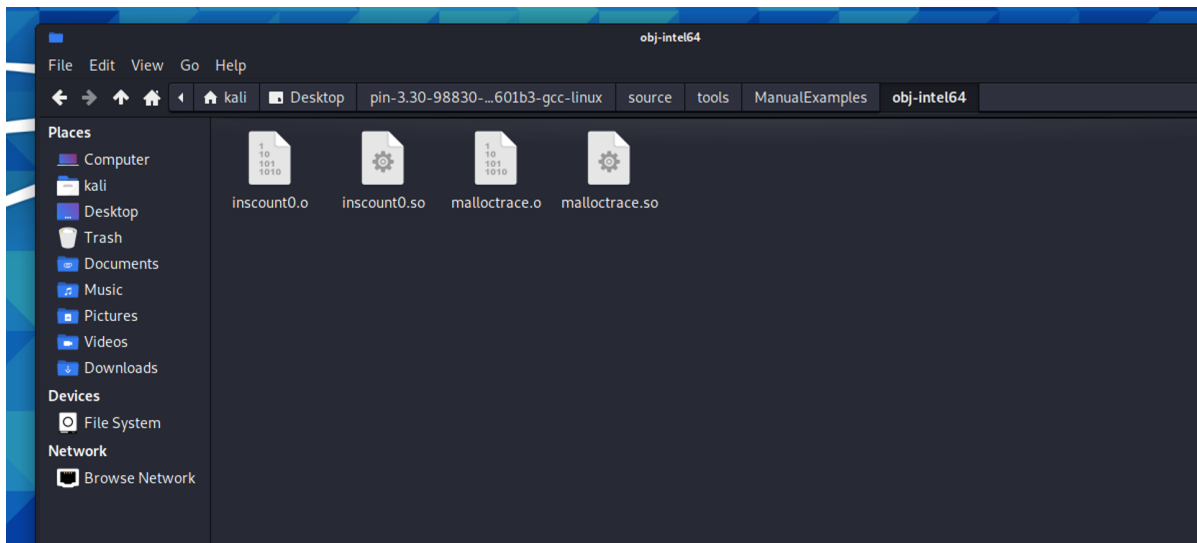


2.2 编译malloctrace.cpp产生动态链接库

我们按照实验要求，打开 `source/tools/ManualExamples`，打开 `malloctrace.cpp` 进行查看源码：



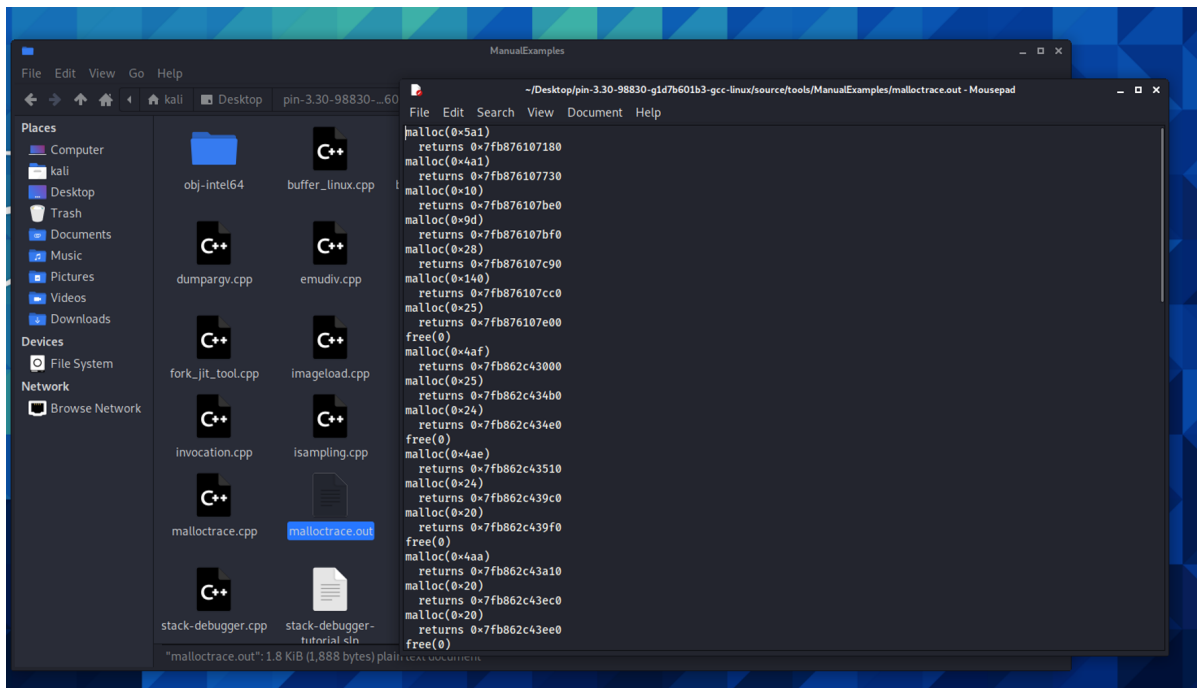
我们输入命令行 `make malloctrace.test TARGET=intel64`，编译运行该文件，得到以下的结果：



我们看到其中产生了 `malloctrace.so` 文件，即成功产生了动态链接库。

2.3 进行插桩实验

在进行完以上的步骤之后，我们就可以应用这个 `tool` 来进行插桩实验了！我们上一部分已经完成了对文件的编译，我们直接使用上面的编译生成的 `out` 文件进行查看，发现确实生成了一大串文字！我们打开文件，查看里面的内容，发现其输出了 `malloc` 和 `free` 函数的具体数值。



我们发现输出的文字由 `malloc` 函数、`returns`、`free` 函数构成，我们在下一个部分我们具体进行介绍。可以发现，每次使用 `malloc` 申请堆内存空间的时候，都会输出申请的空间的大小，并且得到申请的空间的起始地址。当使用 `free` 释放内存空间的时候，会输出释放的空间的起始地址。

3 malloc和free函数输入输出的理解

我们根据上面的输出内容来进一步分析。

```
malloc(0x5a1)
  returns 0x7fb876107180
malloc(0x4a1)
  returns 0x7fb876107730
malloc(0x10)
  returns 0x7fb876107be0
malloc(0x9d)
  returns 0x7fb876107bf0
malloc(0x28)
  returns 0x7fb876107c90
malloc(0x140)
  returns 0x7fb876107cc0
malloc(0x25)
  returns 0x7fb876107e00
free(0)
malloc(0x4af)
  returns 0x7fb862c43000
malloc(0x25)
  returns 0x7fb862c434b0
malloc(0x24)
  returns 0x7fb862c434e0
free(0)
malloc(0x4ae)
  returns 0x7fb862c43510
malloc(0x24)
  returns 0x7fb862c439c0
malloc(0x20)
  returns 0x7fb862c43ec0
malloc(0x20)
  returns 0x7fb862c43ee0
free(0)
```

```
malloc(0x20)
    returns 0x7fb862c439f0
free(0)
malloc(0x4aa)
    returns 0x7fb862c43a10
malloc(0x20)
    returns 0x7fb862c43ec0
malloc(0x20)
    returns 0x7fb862c43ee0
free(0)
malloc(0x4aa)
    returns 0x7fb862c43f00
malloc(0x20)
    returns 0x7fb862c443b0
malloc(0x58)
    returns 0x7fb862c443d0
malloc(0x28)
    returns 0x7fb862c44430
malloc(0x28)
    returns 0x7fb862c44460
malloc(0x38)
    returns 0x7fb862c44490
malloc(0x68)
    returns 0x7fb862c444d0
malloc(0xd8)
    returns 0x7fb862c44540
malloc(0x48)
    returns 0x7fb862c44620
malloc(0x5b8)
    returns 0x7fb862c44670
malloc(0x180)
    returns 0x7fb862c44c30
malloc(0x348)
    returns 0x7fb8625ab000
malloc(0x1b0)
    returns 0x7fb8625ab350
malloc(0x90)
    returns 0x7fb8625ab500
malloc(0x420)
    returns 0x7fb8625ab590
malloc(0x1088)
    returns 0x7fb8625ab9b0
malloc(0x120)
    returns 0x7fb8625aca40
malloc(0x3e0)
    returns 0x7fb8625acb60
malloc(0x7a0)
    returns 0x7fb8624a7000
free(0x7fb8625acb60)
malloc(0xfe0)
    returns 0x7fb8624a77a0
free(0x7fb8624a7000)
malloc(0x11c00)
malloc(0x11c00)
malloc(0x11c00)
    returns 0x1c262a0
```

```

malloc(0x208)
malloc(0x208)
    returns 0x1c37eb0
malloc(0x1d8)
    returns 0x1c380c0
malloc(0x2000)
malloc(0x2000)
    returns 0x1c382a0
malloc(0x200)
malloc(0x200)
    returns 0x1c3a2b0
malloc(0x1d8)
    returns 0x1c3a4c0
malloc(0x2000)
malloc(0x2000)
    returns 0x1c3a6a0
free(0x1c382a0)
free(0x1c382a0)
free(0x1c380c0)
free(0x1c3a6a0)
free(0x1c3a6a0)
free(0x1c3a4c0)

```

首先，前面的 `malloc` 都是系统读入程序后，进行的默认的内存空间的分配，括号中的内容应该是内存空间的大小，`returns` 的值应该是内存进行存储的地址。然后，在使用完内存空间之后，程序会使用 `free` 函数进行释放，注意，释放的时候是读入内存，进行对应位置的内存释放即可。

4 Pintool基本框架

我们在此处分析一下 `Pintool` 的基本框架与源代码。

首先是 `malloctrace` 的分析。

```

#include "pin.H"
#include <iostream>
#include <fstream>
using std::cerr;
using std::endl;
using std::hex;
using std::ios;
using std::string;
std::ofstream TraceFile;

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

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

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

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);
}
}

VOID Fini(INT32 code, VOID* v) { TraceFile.close(); }

INT32 Usage()
{
    cerr << "This tool produces a trace of calls to malloc." << endl;
    cerr << endl << KNOB_BASE::StringKnobSummary() << endl;
    return -1;
}

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

    // Write to a file since cout and cerr maybe closed by the application
    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;
}

```

两个分析函数是 `Arg1Before` 和 `MallocAfter`。`Arg1Before` 在调用 `malloc` 或 `free` 之前执行，用于记录函数名称和参数。`MallocAfter` 在调用 `malloc` 之后执行，记录返回值。`Image` 函数是用于在程序映像中找到并插桩 `malloc` 和 `free` 函数的代码。它会在程序加载时被调用。对于找到的 `malloc` 和 `free` 函数，代码使用 `RTN_InsertCall` 在函数调用前后插入分析函数（如 `Arg1Before` 和 `MallocAfter`）。`Fin` 函数在 `Pin` 工具完成后关闭输出文件。`Usage` 函数提供了一个帮助信息，当命令行参数有误时显示。

以上是该程序的 `PinTool` 的基本框架，下面简单说明一下一般的 `PinTool` 框架：

- 首先要进行初始化，通过调用函数 `PIN_Init` 实现
- 注册插桩函数。通过使用 `XXX_AddInstrumentFunction` 注册一个插桩函数，由于使用了指令级插桩，因此在原始程序的每条指令执行前，都会进入到我们注册的插桩函数中，然后执行相应的操作。
- 注册退出回调函数。通过使用 `PIN_AddFiniFunction` 注册一个程序退出时的回调函数，当应用退出的时候会调用该函数。
- 启动程序。使用函数 `PIN_StartProgram` 启动程序。

心得体会：

通过本次实验，我简单了解了 *Pin* 插桩工具使用的核心步骤，简单总结如下：

1. 编写要进行程序插桩测试的程序，然后编译程序生成可执行文件。
2. 找到要使用的 `pinTool`，将其进行编译，生成可执行的动态链接库。
3. 通过命令 `./pin -t` 命令来进行插桩实验，把 `pinTool` 插入要执行的程序，即可成功实现插桩操作。在对应的目录下会产生 `pinTool` 产生的输出文件。

在本次实验中，我对于linux命令行编辑的使用更加熟练了，在本学期的《计算机组成原理》中也使用了linux编译，也很好的完成了那个实验。