- Pin
  - intel pin 基本用法
    - ❶ 安装Pin
      - 1.通过make命令构建Pintool
        cd source/tools/ManualExamples
        make all TARGET=intel64
    - ❷ Pin和Pintool
      - 2.通过pin命令使用Pintool
        ../../../pin -t obj-intel64/inscount0.so -o inscount0.log -- /bin/ls
      - 3.pin的使用方式为:
        pin [OPTION] [-t <tool> [<toolargs>]] -- < command line>
    - ❸ 插桩分析基本流程
      - 确定需要插装的代码的机制
      - 插桩之后需要执行的分析代码
    - ❹ Pintool基本框架
      - 样例：inscount0.cpp
      - 1.调用PIN_Init(argc, argv)初始化
      - 2.使用INS_AddInstrumentFunction(Instruction, 0)注册一个插桩函数
      - 3.使用PIN_AddFiniFunction(Fini, 0)注册了一个程序退出时的函数
      - 4.使用PIN_StartProgram()启动程序了
      - 5.回调函数模式
        - 1.pintool在编写中将比较多的使用回调函数的机制，譬如在每条指令之前回调Instruction函数
        - 2.在Instruction函数的内部又使用INS_InsertCall注册了一个函数docount，意为在指令执行之前插入一个对docount函数的调用
        - 3.INS_InsertCall是一个变参函数，前3个参数分别为指令，插入的时机（这里IPOINT_BEFORE表示之前）以及函数指针（转为AFUNPTR类型），在之后就可以指定传给函数的参数，并以IARG_END结尾，这里没有指定参数，直接调用。
        - 4.docount的作用即是将一个全局变量加1，以达到统计执行指令条数的目的
        - 回调(diao)函数
          - 定义
            - 1.回调函数就是一个参数，将这个函数作为参数传到另一个函数里面，当那个函数执行完之后，再执行传进去的这个函数。这个过程就叫做回调
            - 2.回调，就是回头调用的意思。主函数的事先干完，回头再调用传进来的那个函数。
          - 例子
            - 定义主函数的时候，我们让代码先去执行callback()回调函数，但输出结果却是后输出回调函数的内容。这就说明了主函数不用等待回调函数执行完，可以接着执行自己的代码。所以一般回调函数都用在耗时操作上面。比如ajax请求，比如处理文件等。

```
//定义主函数，回调函数作为参数
function A(callback) {
    callback();
    console.log('我是主函数');
}

//定义回调函数
function B(){
    setTimeout("console.log('我是回调函数')", 3000);//模仿耗时操作
}

//调用主函数，将函数B传进去
A(B);

//输出结果
我是主函数
我是回调函数
```

    - ❺ 指令级别的插桩
      - 1.指定插桩的位置
        - 1.最简单的情况是直接针对所有指令插桩，INS模块中提供了很多API来判断当前指令的类型

```
INS_IsMemoryRead (INS ins)
INS_IsMemoryWrite (INS ins)
INS_IsLea (INS ins)
INS_IsNop (INS ins)
INS_IsBranch (INS ins)
INS_IsDirectBranch (INS ins)
INS_IsDirectCall (INS ins)
INS_IsDirectBranchOrCall (INS ins)
INS_IsBranchOrCall (INS ins)
INS_IsCall (INS ins)
INS_IsRet (INS ins)
...
```

        - 2.通过加判断条件来对目标类型指令插桩

```
if (INS_Opcode(ins) == XED_ICLASS_MOV &&
    INS_IsMemoryRead(ins) &&
    INS_OperandIsReg(ins, 0) &&
    INS_OperandIsMemory(ins, 1))
```

          - 上面的代码来自safecopy.cpp，直接通过Opcode来识别mov指令，并且是一条内存读指令，并且指令的第一个操作数是寄存器，并且指令的第二个操作数是内存。通过组合这些API就可以非常精确地筛选出想要插桩的指令了。
      - 2.插桩分析代码

```
// This function is called before every instruction is executed
// and prints the IP
VOID printip(VOID *ip) { fprintf(trace, "%p\n", ip); }

// Pin calls this function every time a new instruction is encountered
VOID Instruction(INS ins, VOID *v)
{
    // Insert a call to printip before every instruction, and pass it the IP
    INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)printip, IARG_INST_PTR, IARG_END);
}
```

        - 在这里传递给printip的是一个IARG_INST_PTR参数，实际对应的类型是VOID *，指示了当前指令的位置，而printip则是把它输出出来，所以itrace的作用即是输出所有指令的地址。
        - 实际来说Instrumentation arguments中给出了很多可以传递给回调函数的参数，包括当前指令读取的有效内存地址、相关寄存器的值等等，能够对程序的运行状态有很全面的描述，便于回调函数的进一步分析
    - ❻ 程序运行状态监控 & 修改
      - 1.寄存器
        - 1.获取当前某期存器的值
        - 2.修改当前某寄存器的值
      - 2.内存
        - 1.可以参考SafeCopy，其中使用到了PIN_SafeCopy函数
        - 2.safecopy实际模拟了mov指令内存读的过程，将寄存器和指令操作的内存地址传递给分析函数DoLoad，并在最后用IARG_RETURN_REGS指定将分析函数的返回值写入到指令的操作寄存器中，实际指令的语义没有改变。
        - 3.而在DoLoad函数中，实际调用了PIN_SafeCopy(&value, addr, sizeof(ADDRINT));将对应地址的内容模拟装载并返回。由此就可以看出在程序实际运行时pintool和原始程序位于同一地址空间，因而PIN_SafeCopy既可以从内存中读取数据，亦可以写入数据。
    - ❼ 更粗粒度的插桩
      - 有时我们并不需要在指令级的插桩，pin也可以实现基于Basic Block，Routine或Image的插桩函数，以例子中的malloctrace来说
      - 使用IMG_AddInstrumentFunction来注册一个在Image载入时插桩的函数，随后在Image里面使用RTN_FindByName来找到模块里的malloc和free两个符号，注意在pintool开头除了PIN_Init之外还要用PIN_InitSymbols来初始化symbol manager。在找到相应的函数之后，可以使用RTN_InsertCall来插入分析代码Arg1Before，并将此时函数的参数传递给分析函数。最后这个pintool完成的作用就是追踪malloc/free的调用，并输出它们的参数与返回值。
    - 小结
      - 使用Pin工具需要首先理解二进制程序插桩的过程和整体思路，之后编写pintool就是套用例子就可以了，如果有需要的功能可以直接查手册或者自己去尝试。
  - 分支主题 2
  - 分支主题 3