REVERSE TOOLS: PIN

C.K. Chen(Bletchley)

https://hackmd.io/GYFgJgzA7ApgjAVgLSwIYiSYMMCMAcIuSqCADAMa5gJRzBn5A===

VM ENVIRONMENT

- Environment
  - Ubuntu 14.04
    - Docker + Triton
    - Vxcon/vxcon
- docker
  - sudo docker exec –it vxcon /bin/bash
  - sudo docker run –it vxcon /bin/bash
- Inside the docker
  - Pin
  - Triton
    - /home/vxcon/pin-2.14-71313-gcc.4.4.7-linux/

DYNAMIC BINARY INSTRUMENTATION

- A technique that inserts extra code into a program to collect runtime information.
  - Program analysis : performance profiling, error detection, capture & replay
  - Architectural study : processor and cache simulation, trace collection
  - Binary translation : Modify program behavior, emulate unsupported instructions

INSTRUMENTATION APPROACHES

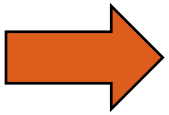
- Source Code Instrumentation (SCI)
  - – instrument source programs

- Binary Instrumentation (BI)
  - instrument binary executable directly

SCI EXAMPLE (CODE COVERAGE)

Original Program

```
void foo() {
  bool found=false;
  for (int i=0; i<100; ++i) {
          if (i==50) break;
          if (i==20) found=true;
  }
  printf("foo\n");
}
```



Instrumented Program

```
char inst[5];
void foo() {
  bool found=false; inst[0]=1;
  for (int i=0; i<100; ++i) {
          if (i==50) { inst[1]=1;break;}
if (i==20) { inst[2]=1;found=true;}
          inst[3]=1;
  }
printf("foo\n");
inst[4]=1;
}
```

BINARY INSTRUMENTATION (BI)

- Static binary instrumentation – inserts additional code and data before execution and generates a persistent modified executable

- Dynamic binary instrumentation – inserts additional code and data during execution without making any permanent modifications to the executable.

ADVANTAGES

- Binary instrumentation
  - Language independent
  - Machine-level view
  - Instrument legacy/proprietary software
- Dynamic instrumentation
  - No need to recompile or relink
  - Discover code at runtime
  - Handle dynamically-generated code
  - Attach to running processes

PIN

- Intel *framework* for creating dynamic analysis tools
- Allows writing fancy tools on x86 and x86_64
- Kind of like scripting a debugger, but less invasive
- Download pin from https://software.intel.com/en-us/articles/pintool-downloads

PIN TOOL

- Build "pin-tools" - programs that run with pin
  - Run in address space of target program
  - Just-in-time compiles target
  - Add in hooks (like breakpoints) for certain events

USING PIN

- Launch and instrument an application

```
$ pin -t pintool.so -- application
```

Instrumentation engine

(provided in the kit)

Instrumentation tool

(write your own, or use one provided in the kit)

Attach to and instrument an application

```
$ pin -t pintool.so -pid 1234
```

PIN AND PINTOOLS

- Pin – the instrumentation engine
- Pintool – the instrumentation program

- Pin provides the framework and API, Pintools run on Pin to perform meaningful tasks.

- Pintools
  - Written in C/C++ using Pin APIs
  - Many open source examples provided with the Pin kit
  - Certain Do's and Don'ts apply

WRITE PIN TOOL

- Main Function

```
int main(int argc, char * argv[])
{
    trace = fopen("itrace.out", "w");

    // Initialize pin
    if (PIN_Init(argc, argv)) return Usage();

    // Register Instruction to be called to instrument instructions
    INS_AddInstrumentFunction(Instruction, 0);

    // Register Fini to be called when the application exits
    PIN_AddFiniFunction(Fini, 0);

    // Start the program, never returns
    PIN_StartProgram();

    return 0;
}
```

MAIN FUNCITON

- PIN_Init() initializes pin
- PIN_AddFiniFunction() add callback when program exits
- PIN_StartProgram() start the program and never return

- *_AddInstrumentFunction insert the instrument code when the first time code executed
  - INS_AddInstrumentFunction: Instruction level
  - TRACE_AddInstrumentFunction: Block level
  - RTN_AddInstrumentFunction: Function level

INSTRUMENT FUNCTION

- Define instrumentation function
  - Insert analysis function
  - Define argument type
- Analysis function – the code you write

```
// 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);
}
```

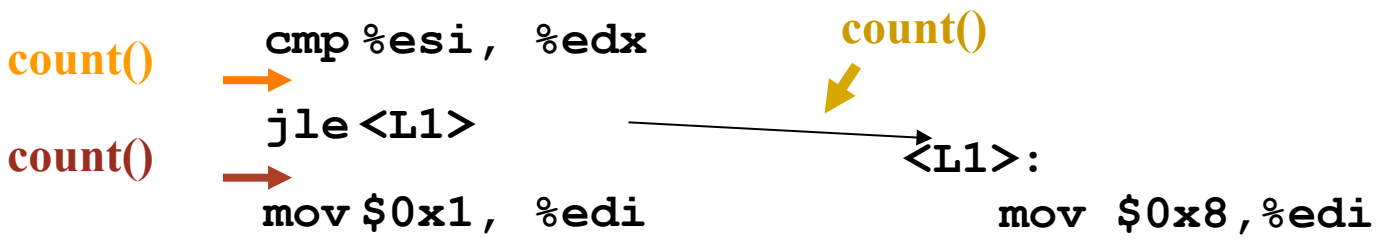
EXAMPLES OF ARGUMENTS TO ANALYSIS ROUTINE

- `IARG_INST_PTR`
  - **Instruction pointer (program counter) value**
- `IARG_UINT32 <value>`
  - **An integer value**
- `IARG_REG_VALUE <register name>`
  - **Value of the register specified**
- `IARG_BRANCH_TARGET_ADDR`
  - **Target address of the branch instrumented**
- `IARG_MEMORY_READ_EA`
  - **Effective address of a memory read**

And many more … (refer to the Pin manual for details)

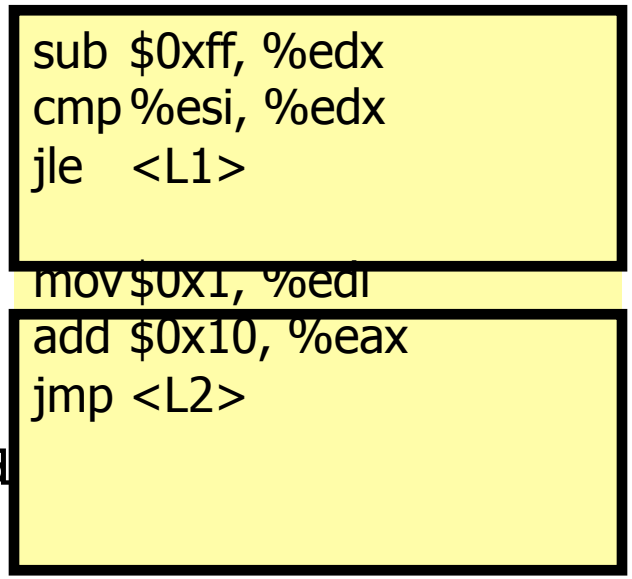
INSTRUMENTATION POINTS

- Instrument points relative to an instruction:
  - *Before (IPOINT_BEFORE)*
  - After:
    - Fall-through edge (IPOINT_AFTER)
    - **Taken edge (IPOINT_TAKEN)**



INSTRUMENTATION GRANULARITY

- **Instruction**
- **Basic block**
  - A sequence of instructions terminated at a control-flow changing instruction
  - Single entry, single exit
- **Trace**
  - A sequence of basic blocks terminated at an unconditional control-flow changing instruction
  - Single entry, multiple exits



1 Trace, 2 BBs, 6 insts

jumpmix example – which types of jump instructions are called?

COMPILE PIN TOOL

- You can write your tool in source/tools/ ManualExamples

```
TEST_TOOL_ROOTS := inscount0 inscount1 inscount2 proccount imageload staticcount detach malloctrace \
                   malloc_mt inscount_tls stack-debugger pinatrace itrace isampling safecopy invocation \
                   countreps nonstatica mycount
```

- Execute make
- P.s. you may need to execute "echo 0 > /proc/sys/kernel/yama/ ptrace_scope"

RUN PINTOOL

- Execute command

```
dsns@ubuntu32:~/pin/pin-2.14-71313-gcc.4.4.7-linux/source/tools/ManualExamples$ ../../../pin -t obj-ia32/inscount0.so -- /bin/ls
buffer_linux.cpp          follow_child_app1.cpp  inscount2.cpp     makefile.rules   proccount.cpp                          staticcount.cpp
buffer_windows.cpp        follow_child_app2.cpp  inscount.out      malloc_mt.cpp    replacesigprobed.cpp                   strace.cpp
countreps.cpp             follow_child_tool.cpp  inscount_tls.cpp  malloctrace.cpp  safecopy.cpp                           thread_unix.c
detach.cpp                fork_app.cpp           invocation.cpp    mycount.cpp      stack-debugger.cpp                     thread_win.c
divide_by_zero_linux.c    fork_jit_tool.cpp      isampling.cpp     nonstatica.cpp   stack-debugger-tutorial.sln            w_malloctrace.cpp
divide_by_zero_windows.c  imageload.cpp          itrace.cpp        obj-ia32         stack-debugger-tutorial.vcxproj
emudiv.cpp                inscount0.cpp          little_malloc.c   pinatrace.cpp    stack-debugger-tutorial.vcxproj.filters
fibonacci.cpp             inscount1.cpp          makefile          pin.log          statica.cpp
dsns@ubuntu32:~/pin/pin-2.14-71313-gcc.4.4.7-linux/source/tools/ManualExamples$ cat inscount.out
Count 850693
dsns@ubuntu32:~/pin/pin-2.14-71313-gcc.4.4.7-linux/source/tools/ManualExamples$
```

PRACTICE: ITRACE

- Implement a simple pin tool to trace every instruction executed

PRACTICE: ITRACE PINTOOL

- Pin tool for instruction trace
  - This practice is the same as itrace.cpp

```
39  int main(int argc, char * argv[])
40  {
41      trace = fopen("itrace.out", "w");
42
43      // Initialize pin
44      if (PIN_Init(argc, argv)) return Usage();
45
46      // Register Instruction to be called to instrument
47      INS_AddInstrumentFunction(Instruction, 0);
48
49      // Register Fini to be called when the application exits
50      PIN_AddFiniFunction(Fini, 0);
51
52      // Start the program, never returns
53      PIN_StartProgram();
54
55      return 0;
56  }
```

```
1  #include <stdio.h>
2  #include "pin.H"
3
4  FILE * trace;
5
```

PRACTICE: MAIN CODE

```
8   VOID printip(VOID *ip) { fprintf(trace, "%p\n", ip); }
9
10  // Pin calls this function every time a new instruction is encountered
11  VOID Instruction(INS ins, VOID *v)
12  {
13      // Insert a call to printip before every instruction, and pass it the IP
14      INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)printip, IARG_INST_PTR, IARG_EN
15  }
16
17  // This function is called when the application exits
18  VOID Fini(INT32 code, VOID *v)
19  {
20      fprintf(trace, "#eof\n");
21      fclose(trace);
22  }
23
24  /* ===================================================================== */
25  /* Print Help Message                                                    */
26  /* ===================================================================== */
27
28  INT32 Usage()
29  {
30      PIN_ERROR("This Pintool prints the IPs of every instruction executed\n"
31                + KNOB_BASE::StringKnobSummary() + "\n");
32      return -1;
33  }
```

PRACTICE: MAKE

- makefile.rules

```
16
17  # This defines tests which run tools of the same name.  This is simply for convenience to avoid
18  # defining the test name twice (once in TOOL_ROOTS and again in TEST_ROOTS).
19  # Tests defined here should not be defined in TOOL_ROOTS and TEST_ROOTS.
20  TEST_TOOL_ROOTS := inscount0 inscount1 inscount2 proccount imageload staticcount detach malloctrace \
21                     malloc_mt inscount_tls stack-debugger pinatrace itrace isampling safecopy invocation \
22                     countreps nonstatica unpack cfi cfi_check instruction
```

- make

RESULT

```
1  7C92118A
2  7C92118C
3  7C92118D
4  7C92118E
5  7C92118F
6  7C921190
7  7C93C4DA
8  7C93C4DD
9  7C93C4E0
10 7C93C529
11 7C93C52F
12 7C93C530
13 7C9211DD
14 7C9211DF
15 7C9211E0
16 7C9211E2
17 7C9211E5
18 7C9211E6
19 7C9211EC
20 7C9211F3
21 7C9211F6
22 7C9211F9
23 7C9211FF
24 7C921203
25 7C921209
26 7C92120A
27 7C92120B
28 7C93C535
29 7C93C4E5
30 7C93C4E9
```

PRACTICE: STRACE

System Call Tracer

- Implement a system call tracer
  - Print out system call number and it's arguments

PRACTICE: CODE(1)

```
90  INT32 Usage()
91  {
92      PIN_ERROR("This tool prints a log of system calls"
93                + KNOB_BASE::StringKnobSummary() + "\n");
94      return -1;
95  }
96
97  /* ===================================================================== */
98  /* Main                                                                  */
99  /* ===================================================================== */
100
101 int main(int argc, char *argv[])
102 {
103     if (PIN_Init(argc, argv)) return Usage();
104
105     trace = fopen("strace.out", "w");
106
107     INS_AddInstrumentFunction(Instruction, 0);
108     PIN_AddSyscallEntryFunction(SyscallEntry, 0);
109     PIN_AddSyscallExitFunction(SyscallExit, 0);
110
111     PIN_AddFiniFunction(Fini, 0);
112
113     // Never returns
114     PIN_StartProgram();
115
116     return 0;
117 }
118
```

```
5   #include <stdio.h>
6
7   #if defined(TARGET_MAC)
8   #include <sys/syscall.h>
9   #elif !defined(TARGET_WINDOWS)
10  #include <syscall.h>
11  #endif
12
13  #include "pin.H"
14
15
16  FILE * trace;
17
```

PRACTICE: CODE(2)

```
58  VOID Instruction(INS ins, VOID *v)
59  {
60      // For O/S's (Mac) that don't support PIN_AddSyscallEntryFunction(),
61      // instrument the system call instruction.
62
63      if (INS_IsSyscall(ins) && INS_HasFallThrough(ins))
64      {
65          // Arguments and syscall number is only available before
66          INS_InsertCall(ins, IPOINT_BEFORE, AFUNPTR(SysBefore),
67                         IARG_INST_PTR, IARG_SYSCALL_NUMBER,
68                         IARG_SYSARG_VALUE, 0, IARG_SYSARG_VALUE, 1,
69                         IARG_SYSARG_VALUE, 2, IARG_SYSARG_VALUE, 3,
70                         IARG_SYSARG_VALUE, 4, IARG_SYSARG_VALUE, 5,
71                         IARG_END);
72
73          // return value only available after
74          INS_InsertCall(ins, IPOINT_AFTER, AFUNPTR(SysAfter),
75                         IARG_SYSRET_VALUE,
76                         IARG_END);
77      }
78  }
79
80  VOID Fini(INT32 code, VOID *v)
81  {
82      fprintf(trace,"#eof\n");
83      fclose(trace);
84  }
```

PRACTICE: CODE(3)

```
19  // Print syscall number and arguments
20  VOID SysBefore(ADDRINT ip, ADDRINT num, ADDRINT arg0, ADDRINT arg1, ADDRINT arg2, ADDRINT arg3, ADDRINT arg4, ADDRINT arg5)
21  {
22      fprintf(trace,"0x%lx: %ld(0x%lx, 0x%lx, 0x%lx, 0x%lx, 0x%lx, 0x%lx)",
23          (unsigned long)ip,
24          (long)num,
25          (unsigned long)arg0,
26          (unsigned long)arg1,
27          (unsigned long)arg2,
28          (unsigned long)arg3,
29          (unsigned long)arg4,
30          (unsigned long)arg5);
31  }
32
33  // Print the return value of the system call
34  VOID SysAfter(ADDRINT ret)
35  {
36      fprintf(trace,"returns: 0x%lx\n", (unsigned long)ret);
37      fflush(trace);
38  }
```

PRACTICE: CODE(4)

```
40  VOID SyscallEntry(THREADID threadIndex, CONTEXT *ctxt, SYSCALL_STANDARD std, VOID *v)
41  {
42      SysBefore(PIN_GetContextReg(ctxt, REG_INST_PTR),
43          PIN_GetSyscallNumber(ctxt, std),
44          PIN_GetSyscallArgument(ctxt, std, 0),
45          PIN_GetSyscallArgument(ctxt, std, 1),
46          PIN_GetSyscallArgument(ctxt, std, 2),
47          PIN_GetSyscallArgument(ctxt, std, 3),
48          PIN_GetSyscallArgument(ctxt, std, 4),
49          PIN_GetSyscallArgument(ctxt, std, 5));
50  }
51
52  VOID SyscallExit(THREADID threadIndex, CONTEXT *ctxt, SYSCALL_STANDARD std, VOID *v)
53  {
54      SysAfter(PIN_GetSyscallReturn(ctxt, std));
55  }
```

PRACTICE: CODE(5)

- makefile.rules

```
16
17  # This defines tests which run tools of the same name.  This is simply for convenience to avoid
18  # defining the test name twice (once in TOOL_ROOTS and again in TEST_ROOTS).
19  # Tests defined here should not be defined in TOOL_ROOTS and TEST_ROOTS.
20  TEST_TOOL_ROOTS := inscount0 inscount1 inscount2 proccount imageload staticcount detach malloctrace \
21                     malloc_mt inscount_tls stack-debugger pinatrace itrace isampling safecopy invocation \
22                     countreps nonstatica unpack cfi cfi_check instruction
23
```

- make

PRACTICE: ICOUNT

Instruction Count Side Channel

- Implement a pin tool which can count the number of executed instructions

PRACTICE: SIDECHANEL

- Try length
  - a, aa , aaa, aaaa, ….
- Try each character
  - a_____, b_____, c_____
  - ba____, bb_____, bc____

Q&A