Execution Tracing w/ Dynamic Binary Instrumentation

1 Lab Overview

One of the greatest dynamic binary instrumentation (DBI) tools is the Intel PIN [1]. It is very easy to learn, and it can be used for many of our program analysis needs.

With PIN, you can nearly observe everything regarding a program execution, e.g., how the program is loaded, where the code/data section gets mapped, when a function gets called/returned, what's the parameters to a function, when a system call is invoked, and what's arguments and return values associated to the system call, etc. While PIN engine itself is not open source, PIN tool suite has tons of examples. The easiest way to learn PIN is to read the manuals (`https://software.intel.com/sites/landingpage/pintool/docs/71313/Pin/html/index.html`). Students are encouraged to play with these examples, and modify them for your needs.

The goal of this project is for the students to get the hands-on experience on dynamic binary instrumentation i.e. analyzing behavior of a binary at runtime by injecting instrumentation code by implementing simple execution tracing tool.

2 Lab Environment

2.1 Lab Setup

This lab requires a virtual machine installed in your environment. You can use the VM at `https://drive.google.com/open?id=0Bwy4q0s779ZcU1JtMXI2UlBNYk0`. You will need to download and install PIN tool.

Note that you can run the VM by using VMware Player (which is free), or Virtual-box (which is open source). You can run this machine with "root" user without password.

Below are the commands to download, install and test the PIN suite.

```
zlin@debian:~$ wget http://software.intel.com/sites/landingpage/pintool/downloads/pin-2.14-71313-gcc.4.4.7-linux.tar.gz \
--no-check-certificate

zlin@debian:~$ tar zxvf pin-2.14-71313-gcc.4.4.7-linux.tar.gz

zlin@debian:~$ ls
pin-2.14-71313-gcc.4.4.7-linux pin-2.14-71313-gcc.4.4.7-linux.tar.gz
zlin@debian:~$ cd pin-2.14-71313-gcc.4.4.7-linux/

zlin@debian:~$ cat README | less

zlin@debian:~$ cd source/tools/SimpleExamples/

zlin@debian:~$ make

zlin@debian:~$ cd obj-32

zlin@debian:~/pin-2.14-71313-gcc.4.4.7-linux/source/tools/SimpleExamples/obj-ia32$ ../../../../pin -t opcodemix.so -- /bin/ls
calltrace.o get_source_app ldstmix.so regval_app
calltrace.so get_source_app_gnu_debug malloctrace.o regval.o
catmix.o get_source_app_gnu_debug.dbg malloctrace.so regval.so
catmix.so get_source_location.o opcodemix.o topopcode.o
coco.o get_source_location.so opcodemix.out topopcode.so
coco.so icount.o opcodemix.so toprtn.o
dcache.o icount.so oper_imm_app toprtn.so
dcache.so ilenmix.o oper_imm_asm.o trace.o
edgcnt.o ilenmix.so oper-imm.o trace.so
edgcnt.so inscount2_mt.o oper-imm.so xed-print.o
emuload.o inscount2_mt.so pinatrace.o xed-print.so
emuload.so jumpmix.o pinatrace.so xed-use.o
```

[1] `https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool`)

```
extmix.o jumpmix.so regmix.o xed-use.so
extmix.so ldstmix.o regmix.so

zlin@debian:~/pin-2.14-71313-gcc.4.4.7-linux/source/tools$ ls
AlignChk Insmix MyPinTool
AttachDetach InstLib pintool.tscons-extension
Buffer InstLibExamples Probes
CacheClient InstructionTests Regvalue
ChildProcess InstrumentationOrderAndVersion Replay
Config JitProfilingApiTests RtnTests
Cpp11Tests MacTests SegmentsVirtualization
CrossIa32Intel64 Maid SegTrace
Debugger makefile SignalTests
DebugInfo ManualExamples SimpleExamples
DebugTrace MaskVector SyncTests
GracefulExit Memory SyscallsEmulation
I18N MemTrace Tests
IArg MemTranslate ToolUnitTests
ImageTests Mix Utils
InlinedFuncsOpt Mmx
```

As you can see from the examples, we have successfully installed PIN, and we are able to execute the plugin opcodemix.so to test the result with "/bin/ls" binary.

3 Examples

Students are encouraged to explore the examples provided in the `pin-2.14-71313-gcc.4.4.7-linux/source/tools/SimpleExamples` and `pin-2.14-71313-gcc.4.4.7-linux/source/tools/ManualExamples` directory to get better understanding of writing code for PIN plugins.

3.1 Understanding code

As shown in `ManualExamples/inscount0.cpp`, if you need to check dynamic instructions executed (the actual number of instructions executed in the binary rather than the instructions present in the program), you can use the following code block.

```
UINT64 ins_count = 0;
KNOB<string> KnobOutputFile(KNOB_MODE_WRITEONCE, "pintool",
    "o", "inscount.out", "specify output file name");
VOID docount()
{
    ins_count++;
}
VOID Instruction(INS ins, VOID *v) {
    // Insert a call to docount before every instruction, no arguments are passed
    INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)docount, IARG_END);
}
VOID Fini(INT32 code, VOID *v)
{
    // Write to a file since cout and cerr maybe closed by the application
    OutFile.setf(ios::showbase);
    OutFile << "Count " << icount << endl;
    OutFile.close();
}
int main(int argc, char *argv[]) {
    INS_AddInstrumentFunction(Instruction, 0);
    PIN_AddFiniFunction(Fini, 0);

    // Never returns
    PIN_StartProgram();
    return 0;
}
```

In the example above, you can see that we need to provide the function we are adding for instrumentation. In our case, this instrumentation function is `Instruction`. This function defines where should we insert our analysis function which in our case is `docount`. We can use multiple `INS_AddInstrumentFunction` functions to add more instrumentation functionality.

Finally, to provide the analysis, we can either use the `stderr` or we could write to file object. Since this should be done at the end of the command execution, we add a finish function to provide this report. This function is added to the instrumentation using `PIN_AddFiniFunction`.

3.2 Understanding how to take traces

If you take a look at `ManualExamples/strace.cpp`, you'll see that we can trace system calls using PIN.

```
// Print syscall number and arguments
VOID SysBefore(ADDRINT ip, ADDRINT num, ADDRINT arg0, ADDRINT arg1, ADDRINT arg2,
    ADDRINT arg3, ADDRINT arg4, ADDRINT arg5)
{
    fprintf(trace,"0x%lx: %ld(0x%lx, 0x%lx, 0x%lx, 0x%lx, 0x%lx, 0x%lx)",
        (unsigned long)ip,
        (long)num,
        (unsigned long)arg0,
        (unsigned long)arg1,
        (unsigned long)arg2,
        (unsigned long)arg3,
        (unsigned long)arg4,
        (unsigned long)arg5);
}
// Print the return value of the system call
VOID SysAfter(ADDRINT ret)
{
    fprintf(trace,"returns: 0x%lx\n", (unsigned long)ret);
    fflush(trace);
}
VOID SyscallEntry(THREADID threadIndex, CONTEXT *ctxt, SYSCALL_STANDARD std, VOID *v)
{
    SysBefore(PIN_GetContextReg(ctxt, REG_INST_PTR),
        PIN_GetSyscallNumber(ctxt, std),
        PIN_GetSyscallArgument(ctxt, std, 0),
        PIN_GetSyscallArgument(ctxt, std, 1),
        PIN_GetSyscallArgument(ctxt, std, 2),
        PIN_GetSyscallArgument(ctxt, std, 3),
        PIN_GetSyscallArgument(ctxt, std, 4),
        PIN_GetSyscallArgument(ctxt, std, 5));
}
VOID SyscallExit(THREADID threadIndex, CONTEXT *ctxt, SYSCALL_STANDARD std, VOID *v)
{
    SysAfter(PIN_GetSyscallReturn(ctxt, std));
}
VOID Fini(INT32 code, VOID *v)
{
    fprintf(trace,"#eof\n");
    fclose(trace);
}
int main(int argc, char *argv[])
{
    trace = fopen("strace.out", "w");

    PIN_AddSyscallEntryFunction(SyscallEntry, 0);
    PIN_AddSyscallExitFunction(SyscallExit, 0);

    PIN_AddFiniFunction(Fini, 0);
```

```
    // Never returns
    PIN_StartProgram();

    return 0;
}
```

This code will run on Linux machines. In this example, we add the syscall entry(`SyscallEntry`) and exit(`SyscallExit`) functions using the PIN functions. The entry function provide the arguments and the syscall number of the system call. The exit function simply provides the return value of the system call. Using these we can create a list of the calls in the chronological order.

4 Lab Tasks

Once you have installed PIN and studied the examples, you need to develop a PIN plugin to trace the binary code execution. In particular, we expect your tool to be able to do the following:

- (**5 Points**) This tool should count how many instructions that have been executed with the testing binary code.
- (**5 Points**) This tool should print what the loaded images are while executing the binary.
- (**20 Points**) This tool should be able to count the number of executions for particular instructions, e.g., MOV, ADD, etc.
- (**20 Points**) The tool should print all function calls/returns, and print the function symbol name if there is any.
- (**20 Points**) The tool should pay special attention to heap management calls (e.g., `malloc/free`) such as print when these function get called, how many bytes allocated, etc.
- (**30 Points**) The tool should recognize all Syscalls, and should be able to interpret a set of well-known system calls with its arguments and return values (look at the output from tool `strace`). Note that you don't have to interpret all system calls, which will be very tedious. But you need to interpret a set of them (e.g., `read, write, open`, etc.). So it's entirely up to you on how many you will interpret. The more you did, the more bonus points you will get. Regarding how to trace system calls in PIN, you can refer the examples in `ToolUnitTests/strace_ia32.cpp`.

Note that it should be a single tool that has the above functionalities.

You can either write your own make file or you can reuse the framework provided in MyPinTool, and modify `MyPinTool.cpp`, as shown below.

```
zlin@debian:~/pin-2.14-71313-gcc.4.4.7-linux/source/tools$ cd MyPinTool/
zlin@debian:~/pin-2.14-71313-gcc.4.4.7-linux/source/tools/MyPinTool$ ls
makefile makefile.rules MyPinTool.cpp
zlin@debian:~/pin-2.14-71313-gcc.4.4.7-linux/source/tools/MyPinTool$ make
zlin@debian:~/pin-2.14-71313-gcc.4.4.7-linux/source/tools/MyPinTool$ ls makefile makefile.rules MyPinTool.cpp obj-ia32
```

5 Submission

You need to submit a report describing your implementation for each requirement. You will also provide the source code for the tool along with `Makefile` for grading.