CS451 – Software Analysis

Lecture 13
Introduction to Intel Pin

Elias Athanasopoulos
athanasopoulos.elias@ucy.ac.cy

Introduction

- Intel Pin is one of the most popular DBI platforms
  - Developed by Intel targeting their CPU family
  - Not open source
- Available for Linux, Windows and OSX
- Pin executes code at the level of *traces*
  - A *trace* is similar to a basic block
  - Defined as an instruction sequence that ends up when it hits an unconditional-transfer instruction or reaches a predefined maximum length or number of unconditional control-flow instructions
  - A trace can be entered only at the top, but has many exits

Implementing Pin tools

- A Pin tool is a program that instruments another program using Pin
  - Distributed in the form of a shared library
  - Developed in C/C++ using the Pin API
- A Pin tool has two basic components
  - Instrumentation routines that tell Pin which instrumentation code to add and where
  - Analysis routines that perform the work based on the collected information

A profiler based on Pin

- A Pin tool that records statistics about a program's execution
  - Number of executed instructions
  - Number of times basic blocks, functions, and syscalls are invoked
- Profiling is at the binary level
- The analyzed program runs using Pin and our Pin tool attached

Setup

- For using the Pin API, you need to include pin.H
- KNOB is a class that allows you to pass command-line options to your tool

```
#include "pin.H"

KNOB<bool> ProfileCalls(KNOB_MODE_WRITEONCE, "pintool",
"c", "0", "Profile function calls");

KNOB<bool> ProfileSyscalls(KNOB_MODE_WRITEONCE, "pintool",
"s", "0", "Profile syscalls");
```

Data structures

- We use C++ data structures to keep the statistics of the profile analysis
- Map addresses (e.g., of a control-flow target) to another map with addresses of the (control-flow) instructions

```
std::map<ADDRINT, std::map<ADDRINT, unsigned long> > cflows;
std::map<ADDRINT, std::map<ADDRINT, unsigned long> > calls;
std::map<ADDRINT, unsigned long> syscalls;
std::map<ADDRINT, std::string> funcnames;

unsigned long insn_count    = 0;
unsigned long cflow_count   = 0;
unsigned long call_count    = 0;
unsigned long syscall_count = 0;§
```

Pin initialization

```
int main(int argc, char *argv[]) {

  PIN_InitSymbols();
  if(PIN_Init(argc,argv)) {
    print_usage();
    return 1;
  }

  /* Register instrumentation functions */

  /* Never returns */
  PIN_StartProgram();

  return 0;
}
```

Registering instrumentation functions

- IMG = image granularity
- INS = instruction granularity
- TRACE = trace granularity

```
/* parse_funcsyms(IMG img, void *v) */
IMG_AddInstrumentFunction(parse_funcsyms, NULL);

/* instrument_insn(INS ins, void *v) */
INS_AddInstrumentFunction(instrument_insn, NULL);

/* instrument_trace(TRACE trace, void *v) */
TRACE_AddInstrumentFunction(instrument_trace, NULL);
```

Registering a syscall entry and a fini function

- Registering a function that executes whenever a system call is entered
  - You can also catch a system call's exit using PIN_ADDSyscallExitFunction()
  - This callback is set only when -s is given as an option to the tool
- A fini function is executing when the PIN session is finished

```
if (ProfileSyscalls.Value()) {
    PIN_AddSyscallEntryFunction(log_syscall, NULL);
}
PIN_AddFiniFunction(print_results, NULL);
```

Parsing function symbols

- Works at the image level, which is the entire executable object
- Loops all over sections to find symbols
- RTN is for routine (i.e., a function)

```
static void parse_funcsyms(IMG img, void *v) {

  if(!IMG_Valid(img)) return;

  for(SEC sec = IMG_SecHead(img); SEC_Valid(sec); sec = SEC_Next(sec)) {
    for(RTN rtn = SEC_RtnHead(sec); RTN_Valid(rtn); rtn = RTN_Next(rtn)) {
      funcnames[RTN_Address(rtn)] = RTN_Name(rtn);
    }
  }
}
```

Instrumenting basic blocks

- The profiler needs to count every instruction that is executed by the process
  - A naïve approach is to add a callback per each instruction, but this is slow
  - A better option is to add a callback per basic block and count the instructions each time a new basic block is processed
- Pin discovers basic blocks on the fly
  - This means that a large basic block may be divided in smaller basic blocks, when Pin executes more code
  - This does not interfere with the profiler's satistics

Adding the basic-block callback

- For Pin basic blocks are called *traces*
- A trace is a basic block that has one entrance but multiple exits

```
static void instrument_trace(TRACE trace, void *v) {
  IMG img = IMG_FindByAddress(TRACE_Address(trace));
  /* Do not count BBs in shared libraries and the dynamic loader.  */
  if(!IMG_Valid(img) || !IMG_IsMainExecutable(img))
        return;

  for(BBL bb = TRACE_BblHead(trace); BBL_Valid(bb); bb = BBL_Next(bb)) {
    instrument_bb(bb);
  }
}
```

Instrumenting the basic block

- The callback is inserted using IPOINT_ANYWHERE, because it does not matter for counting the instructions in the basic block
  - This allows PIN to optimize the placement

```
static void count_bb_insns(UINT32 n) {
  insn_count += n;
}
```

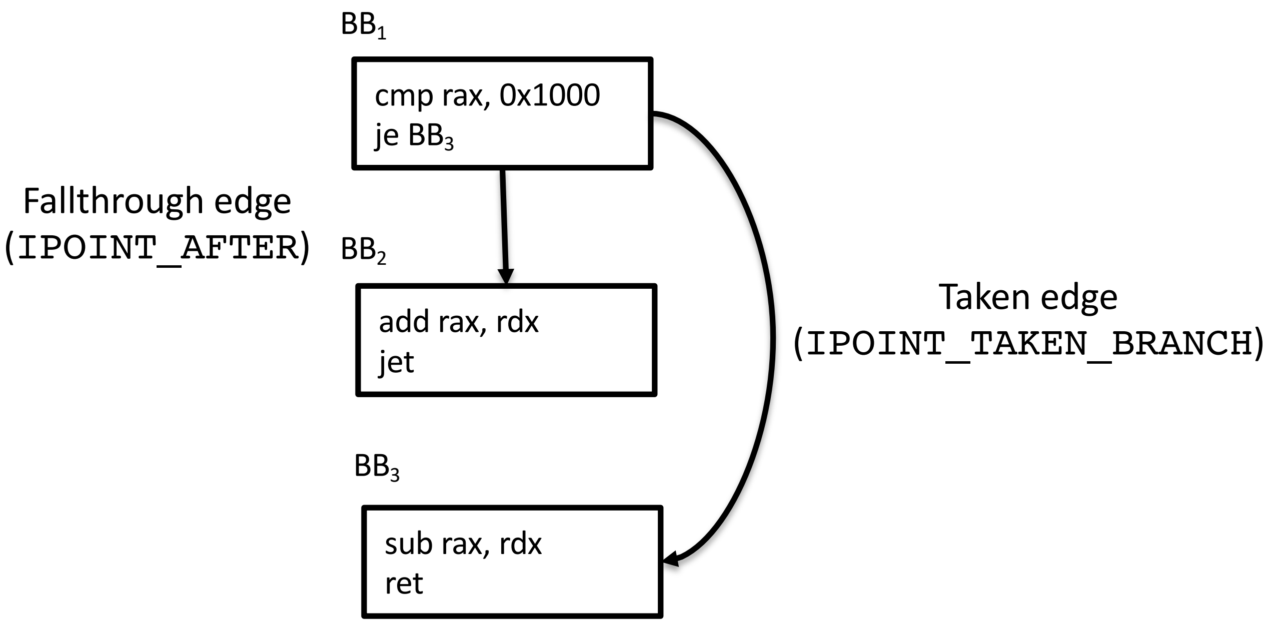
```
static void instrument_bb(BBL bb) {
  BBL_InsertCall(
    bb, IPOINT_ANYWHERE, (AFUNPTR)count_bb_insns,
    /* Data sent to the callback.  */
    IARG_UINT32, BBL_NumIns(bb),
    IARG_END
  );
}
```

Callback placements

Insertion point	Analysis callback	Validity
`IPOINT_BEFORE`	Before instrumented object	Always valid
`IPOINT_AFTER`	On fall-through edge (of branch or "regular" instruction)	If `INS_HasFallThrough` is `true`
`IPOINT_ANYWHERE`	Anywhere in instrumented object	For TRACE or BB only
`IPOINT_TAKEN_BRANCH`	On taken edge or branch	If `INS_IsBranchOrCall` is `true`

Control-flow instructions

- The profiler counts all instructions executed by the process, as well as the number of control-flow transfers and, optionally, the number of calls



Instrumenting control-flow instructions

```
static void instrument_insn(INS ins, void *v) {
  if(!INS_IsBranchOrCall(ins)) return;

  IMG img = IMG_FindByAddress(INS_Address(ins));
  if(!IMG_Valid(img) || !IMG_IsMainExecutable(img)) return;

  INS_InsertPredicatedCall(ins, IPOINT_TAKEN_BRANCH, (AFUNPTR)count_cflow,
    IARG_INST_PTR, IARG_BRANCH_TARGET_ADDR, IARG_END);

  if(INS_HasFallThrough(ins)) {
    INS_InsertPredicatedCall(ins, IPOINT_AFTER, (AFUNPTR)count_cflow,
      IARG_INST_PTR, IARG_FALLTHROUGH_ADDR, IARG_END);
  }

  if(INS_IsCall(ins)) {
    if(ProfileCalls.Value()) {
      INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)count_call,
        IARG_INST_PTR,IARG_BRANCH_TARGET_ADDR, IARG_END);
    }
  }
}
```

Instrumenting the taken edge

- We insert a callback on the instruction’s taken edge
  - We use `INS_InsertPredicatedCall` for calling the callback only when the instruction is actually called
  - Some x86 instructions have built-in conditionals that may prevent their execution (e.g., do a mov operation only when ZF=1)
  - The callback increments the counter of the recorded control-flow instructions (`cflow_count`)
- The callback takes two arguments
  - The value of the instruction pointer (`IARG_INST_PTR`)
  - The address of the target (`IARG_BRANCH_TARGET`)

Instrumenting the fallthrough edge

- If there is a fallthrough edge, we add again the count_cflow() callback
  - In this case, we use IPOINT_AFTER for placing the callback
  - We pass the address of the fallthrough block (`IARG_FALLTHROUGH_ADDR`)

Instrumenting calls

- We use a separate counter for recording calls
- The profiler will record calls only when -c is passed as an argument
- We use `INS_InsertCall` to insert the callback since the `call` instruction has no built-in conditionals

The analysis callbacks

```
static void count_bb_insns(UINT32 n) {
  insn_count += n;
}

static void count_cflow(ADDRINT ip, ADDRINT target) {
  cflows[target][ip]++;
  cflow_count++;
}

static void count_call(ADDRINT ip, ADDRINT target) {
  calls[target][ip]++;
  call_count++;
}

static void log_syscall(THREADID tid, CONTEXT *ctxt, SYSCALL_STANDARD std, VOID *v) {
  syscalls[PIN_GetSyscallNumber(ctxt, std)]++;
  syscall_count++;
}
```

Homework

- Use profiler to gather statistics of existing applications
- Try with a program that you have created
  - With a non stripped and a stripped version
- Add new statistics
  - Count the number of instructions that read from memory
  - Count the number of instructions that write to memory