Hands-on Workbook

Table of Contents

Hands-on Objective

In today's tutorial you will learn to build your first Pintool, modify the tool, as well as debug the Pintool. We will start with a simple "`icount`" Pintool that will evolve from doing instruction counting to tracing instruction and memory addresses. In the process of doing so, we will also learn to debug a Pintool using the system's native debugger. Overall, you will gain insight into the ease of using Pin for program introspection.

In the following exercises, we will step through examples in the Linux environment on a IA32-bit platform. Many of the discussed examples are available for download along with the Pin kit. The kit contains source code for a large number of other example instrumentation tools such as basic block profilers, cache simulators and instruction trace generators. It is easy to derive new tools using the examples as templates.

Overview of Pin

The best way to think about Pin is as a "just in time" (JIT) compiler (a.k.a dynamic compiler). The input to this compiler is not bytecode, but a regular executable. Pin dynamically re-compiles the application during execution. Dynamically compiled code is cached and reused for boosted performance.

Instrumentation of the application is done by the Pin Tool during dynamic compilation. The interface between Pin and the Pin Tool are callback routines. The Pintool registers callbacks, which Pin invokes as it generates new code. This allows the Pintool to decide the instrumentation and analysis “on the fly.” Conceptually, instrumentation consists of (1) deciding where and what code is inserted (instrumentation code) and (2) executing the inserted instrumentation (analysis code). The instrumentation calls target arbitrary functions within the Pintool. Pin ensures that register state is saved and restored as necessary for arguments to be passed to the functions. In addition to pure program introspection, Pin is also capable of altering application behavior.

Instrumentation Granularity

As described above, Pin's instrumentation is "just in time" (JIT). Instrumentation occurs immediately before a code sequence is executed for the first time. We call this mode of operation trace-level instrumentation.

Trace instrumentation lets the Pintool inspect and instrument an executable one dynamic trace of instructions at a time. Pin breaks a trace into basic blocks, BBLs. A BBL is a single entrance, single exit sequence of instructions. It is often possible to insert a single analysis call for a BBL, instead of one analysis call for every instruction (which incurs more runtime overhead). Pin also offers an instruction instrumentation mode which lets the tool inspect and instrument an executable a single instruction at a time. In this tutorial, we will focus on using the instruction instrumentation API’s as they are used frequently, provide most insight and showcase Pin’s rich API.

Sometimes, however, it is useful to look at a larger granularity such as an entire image or entire routine/function. Image instrumentation lets the Pintool inspect and instrument an entire binary, IMG, when it is first loaded. A Pintool can walk the sections, SEC, of the image (.text, .data, .rodata etc.), the routines, RTN, of a section, as well as the instructions, INS contained within a routine. Image instrumentation depends on the presence of symbol information in the executable.

Routine instrumentation lets the Pintool inspect and instrument an entire routine before the first time it is called. A Pintool can walk the instructions of a routine. Instrumentation can be inserted so that it is executed before or after a routine is executed, or before or after an instruction is executed.

Applying a Pintool to an Application

An application and a tool are invoked as follows:

pin [pin-option]... -t [toolname] [tool-options]... -- [application] [application-option]

The following Pin-options are frequently used:

- -t toolname: specifies the Pintool to use.
- -pause_tool n: is a useful Pin-option which prints out the process id and pauses Pin for n seconds to permit attaching with gdb.
- -pid pid: attach Pin and the Pintool to an already running executable with the given process id.

The tool-options follow immediately after the tool specification and depend on the tool used. Everything following the -- is the command line for the application.

Installing and Running Today's Examples

Login: pinhead[0-25]@suburban.colorado.edu
Password: pinme2008!

1) Install a kit by unpacking a kit and changing into the directory:

```
$ tar zxf pin-2.3-17236-gcc.3.4.2-ia32-linux.tar.gz

$ cd pin-2.3-17236-gcc.3.4.2-ia32-linux
```

2) Build the "`pintool.so`" Pintool in the directory:

```
$ cd Tutorial/

$ make pintool.so
```

3) Run the Pintool:

```
$ ../Bin/pin -t ./pintool.so -- /bin/ls

Makefile          atrace.o     imageload.out  itrace      proccount
Makefile.example  imageload    inscount0      itrace.o    proccount.o
atrace            imageload.o  inscount0.o    itrace.out
Count 422838

$
```

Understanding Instruction Counting

Let us take a peek inside the tool to gain an understanding of how to write our own tool. The example we just ran is `pintool.cpp`, which instruments every instruction in the program to count the total number of instructions executed. The Pintool works by inserting an instrumentation call to the analysis function `docount` that increments a counter. The instrumentation call tells Pin to setup instrumentation such that `docount` is called upon instruction execution. In addition, `PIN_AddFiniFunction` tells Pin to call the function `Fini` when the program exits so that the variable can be printed to the screen. Take a moment to understand the detailed comments below:

```
#include <stdio.h>
#include "pin.H"
using namespace std;

// The running count of instructions is kept here
UINT64 icount = 0;

// This function is called before every instruction is executed
VOID docount() { icount++; }

// Pin calls this function every time a new instruction is encountered
VOID Instruction(INS ins, VOID *v)
{
    // Insert a call to docount before every instruction
    INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR) docount, IARG_END);
}

// This function is called when the application exits
VOID Fini(INT32 code, VOID *v)
{
    fprintf(stderr, "%llu\n", icount);
}

// argc, argv are the entire command line, including pin -t <toolname> -- ...
int main(int argc, char * argv[])
{
    // Initialize pin
    PIN_Init(argc, argv);

    // 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;
}
```

Passing Instrumentation Arguments (IARG's)

In the previous example, we did not pass any arguments to the analysis procedure (`docount`). In this example, we will extend the tool to pass arguments. Pin allows us to pass a variety of arguments such as the instruction pointer, current value of registers, effective address of memory operations, constants, etc.

With a small change, we can turn the previous instruction counting example into a Pintool that prints the address of every instruction executed. This tool is useful for understanding the control flow of a program for debugging, or in processor design when simulating an instruction cache.

Expected output of the Pintool:

```
$ ../Bin/pin -t pintool.so -- /bin/ls

Makefile          atrace.o     imageload.out  itrace      proccount
Makefile.example  imageload    inscount0      itrace.o    proccount.o
atrace            imageload.o  inscount0.o    itrace.out

$ head itrace.out

0x40001e90
0x40001e91
0x40001ee4
0x40001ee5
0x40001ee7
0x40001ee8
0x40001ee9
0x40001eea
0x40001ef0
0x40001ee0

$
```

Follow the below steps to modify `pintool.cpp` to perform instruction address tracing:

Step 1: Change the arguments to `INS_InsertCall` to pass the address of the instruction about to be executed. The address of the instruction can be requested from Pin using IARG_INST_PTR.

Step 2: Replace `docount` with `printip`, which prints the instruction address. Note that the analysis function is modified to take a proper argument (`VOID *`).

Step 3: Add file I/O to prevent writing the output to screen.

```
#include <stdio.h>
#include "pin.H"

FILE * trace;

// This function is called before every instruction is executed
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);
}

// This function is called when the application exits
VOID Fini(INT32 code, VOID *v)
{
    fclose(trace);
}

// argc, argv are the entire command line, including pin -t <toolname> -- ...
int main(int argc, char * argv[])
{
    trace = fopen("itrace.out", "w");

    // Initialize pin
    PIN_Init(argc, argv);

    // 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;
}
```

Generic and Portable Inspection API

The previous example instruments all instructions. Sometimes a tool may only want to instrument a class of instructions, like memory operations or branch instructions. A tool can do this by using the Pin API that includes functions that classify and examine instructions.

In this example, we show how to do more selective instrumentation by examining the instructions. This tool generates a trace of all memory addresses referenced by a program. This is useful for debugging and for simulating a data cache in a processor.

We only instrument instructions that read or write memory. In addition, we also make minor modifications to make the Pintool portable across architectures.

Expected output of the Pintool:

```
$ ../Bin/pin -t pintool.so -- /bin/ls

Makefile          atrace.o    imageload.o    inscount0.o  itrace.out
Makefile.example  atrace.out  imageload.out  itrace       proccount
atrace            imageload   inscount0      itrace.o     proccount.o

$ head pinatrace.out

0x40001ee0: R 0xbfffe798
0x40001efd: W 0xbfffe7d4
0x40001f09: W 0xbfffe7d8
0x40001f20: W 0xbfffe864
0x40001f20: W 0xbfffe868
0x40001f20: W 0xbfffe86c
0x40001f20: W 0xbfffe870
0x40001f20: W 0xbfffe874
0x40001f20: W 0xbfffe878
0x40001f20: W 0xbfffe87c
$
```

Follow the below steps to modify `pintool.cpp` to perform memory tracing:

Step 1: Alter your instrumentation routine to inspect only those instructions affecting memory by querying whether an instruction reads or writes memory by using `INS_IsMemoryRead` and `INS_IsMemoryWrite` API calls.

Step 2: Add the required analysis function calls (`RecordMemRead` and `RecordMemWrite`).

Step 3: Modify your instrumentation calls to make the Pintool portable across platforms. Use `INS_InsertPredicatedCall` instead of `INS_InsertCall` to avoid generating references to instructions that are predicated and the predicate is false (predication is only relevant for IA-64 ISA).

Step 4: Extend the instrumentation arguments (IARG's) by adding `IARG_MEMORYREAD_EA` and `IARG_MEMORYWRITE_EA` corresponding to the arguments of the analysis function calls.

Step 5: Change the name of the output file to "pinatrace.out".

```
#include <stdio.h>
#include "pin.H"

FILE * trace;

// Print a memory read record
VOID RecordMemRead(VOID * ip, VOID * addr)
{
    fprintf(trace,"%p: R %p\n", ip, addr);
}

// Print a memory write record
VOID RecordMemWrite(VOID * ip, VOID * addr)
{
    fprintf(trace,"%p: W %p\n", ip, addr);
}

VOID Instruction(INS ins, VOID *v)
{
    // instruments loads using a predicated call, i.e.
    // the call happens iff the load will be actually executed
    // (does not matter for ia32 but arm & ipf have predicated instructions)
    if (INS_IsMemoryRead(ins))
    {
        INS_InsertPredicatedCall(
            ins, IPOINT_BEFORE, (AFUNPTR)RecordMemRead,
            IARG_INST_PTR,
            IARG_MEMORYREAD_EA,
            IARG_END);
    }
```

```
    // instruments stores using a predicated call, i.e.
    // the call happens iff the store will be actually executed
    if (INS_IsMemoryWrite(ins))
    {
        INS_InsertPredicatedCall(
            ins, IPOINT_BEFORE, (AFUNPTR)RecordMemWrite,
            IARG_INST_PTR,
            IARG_MEMORYWRITE_EA,
            IARG_END);
    }
}

// This function is called when the application exits
VOID Fini(INT32 code, VOID *v)
{
    fclose(trace);
}

// argc, argv are the entire command line, including pin -t <toolname> -- ...
int main(int argc, char * argv[])
{
    trace = fopen("pinatrace.out", "w");

    // Initialize pin
    PIN_Init(argc, argv);

    // 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;
}
```

Debugging a Pintool

There are 3 different programs residing in the address space. The application, the Pin instrumentation engine, and your Pintool. This section describes how to use `gdb` to find bugs in a Pintool. You cannot run Pin directly from `gdb` since Pin uses the debugging API to start the application. Instead, you must invoke Pin from the command line with the `-pause_tool` switch, and use `gdb` to attach to the Pin process from another window. The `-pause_tool` switch makes Pin print out the process identifier (`pid`) and pause for `n` seconds.

If your tool is called `pintool.so` and the application is `/bin/ls`, you can use `gdb` as follows:

Start `gdb` with your tool, but do not use the `run` command:

```
$ gdb pintool.so

GNU gdb 5.2.1
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i686-pc-linux-gnu"...
(gdb)
```

In another window, start your application with the `-pause_tool` switch:

```
$ ../Bin/pin -pause_tool 30 -t pintool.so -- /bin/ls

Pausing to attach to pid 28769
```

Then go back to `gdb` and `attach` to the process:

```
(gdb) attach 28769
Attaching to program: .../tutorial/pintool.so, process 28769
0x011ef361 in ?? ()
(gdb)
```

Now, instead of using the `gdb` run command, you use the `cont` command to continue execution. You can also set breakpoints as normal:

```
(gdb) break main
Breakpoint 1 at 0x5048d30: file .../tutorial/pintool.so, line 232.
(gdb) cont
Continuing.

Breakpoint 1, main (argc=6, argv=0x4fef534)
    at .../tutorial/pintool.so:32
(gdb)
```

Detecting the Loading and Unloading of Images (Image Instrumentation)

It is also possible to use Pin to examine binaries without instrumenting them. This is useful when you need to know static properties of an executable. For instance, compiler engineers may want to study the ratio of static code size to the dynamic footprint of an application. Both of these tasks can be achieved using Pin's rich (yet simple) API.

The Pintool on the next page prints a message to a trace file every time and image is loaded or unloaded as show here:

```
$ ../Bin/pin -t ./imageload.so -- /bin/ls

Makefile          atrace.o    imageload.o    inscount0.o  proccount
Makefile.example  atrace.out  imageload.out  itrace       proccount.o
atrace            imageload   inscount0      itrace.o     trace.out

$ cat imageload.out

Loading /bin/ls
Loading /lib/ld-linux.so.2
Loading /lib/libtermcap.so.2
...
Loading /lib/i686/libc.so.6
Unloading /bin/ls
Unloading /lib/ld-linux.so.2
Unloading /lib/libtermcap.so.2
Unloading /lib/i686/libc.so.6
...

$
```

```
#include <stdio.h>
#include "pin.H"

FILE * trace;

// Pin calls this function every time a new img is loaded
// It can instrument the image, but this example does not
// Note that imgs (including shared libraries) are loaded lazily
VOID ImageLoad(IMG img, VOID *v)
{
    fprintf(trace, "Loading %s\n", IMG_Name(img).c_str());
}

// Pin calls this function every time a new img is unloaded
// You can't instrument an image that is about to be unloaded
VOID ImageUnload(IMG img, VOID *v)
{
    fprintf(trace, "Unloading %s\n", IMG_Name(img).c_str());
}

// This function is called when the application exits
// It prints the name and count for each procedure
VOID Fini(INT32 code, VOID *v)
{
    fclose(trace);
}

// argc, argv are the entire command line, including pin -t <toolname> -- ...
int main(int argc, char * argv[])
{
    trace = fopen("imageload.out", "w");

    // Initialize symbol processing
    PIN_InitSymbols();

    // Initialize pin
    PIN_Init(argc, argv);

    // Register ImageLoad to be called when an image is loaded
    IMG_AddInstrumentFunction(ImageLoad, 0);

    // Register ImageUnload to be called when an image is unloaded
    IMG_AddUnloadFunction(ImageUnload, 0);

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

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

    return 0;
}
```

Examining Routines in an Image

It is possible to get insight into the properties of the image loaded into memory by parsing the sections (`.data`, `.rodata` etc.) encoded in the binary, as well as the routines (i.e. functions) contained within the .text section of the executable.

In the following example, we will work with on the `imageload.so` Pintool to make it print the number of instructions contained in every function. The output expected upon running the tool after modifications is show below:

```
$ ../Bin/pin -t ./imageload.so -- /bin/ls

Makefile          atrace.o    imageload.o    inscount0.o  proccount
Makefile.example  atrace.out  imageload.out  itrace       proccount.o
atrace            imageload   inscount0      itrace.o     trace.out

$ cat imageload.out

_init has 8 insts
.plt has 286 insts
.text has 10166 insts
close_stdout has 14132 insts
rpl_free has 14901 insts
_fini has 12 insts
.plt has 19 insts
_dl_sysinfo_int80 has 33 insts
relocate_doit has 49 insts
map_doit has 71 insts
version_check_doit has 102 insts
_dl_initial_error_catch_tsd has 111 insts
rtld_lock_default_lock_recursive has 119 insts
rtld_lock_default_unlock_recursive has 127 insts

$
```

Follow the below steps to print the number of instructions in every routine:

Step 1: Iterate over all sections of the executable using the `SEC_*` API (as shown below).

Step 2: Iterate over all the routines in a section using the shown `RTN_*` API.

Step 3: Add a counter to keep track of the number of instructions in a routine.

Step 4: Request Pin to make all instruction contained within a RTN accessible via `RTN_Open`.

Step 5: Increment the counter while iterating over every instruction in the routine.

Step 6: Release RTN resources to conserve memory, as RTN processing is expensive.

Step 7: Write the collected output to a file.

```
#include <stdio.h>
#include "pin.H"

FILE * trace;

// Pin calls this function every time a new img is loaded
// It can instrument the image, but this example merely
// counts the number of static instructions in the image
VOID ImageLoad(IMG img, VOID *v)
{
    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))
        {
            UINT32 count = 0;

            // Prepare processing of RTN. RTN does not broken up into BBLs,
            // it is merely a sequence of INSs
            RTN_Open(rtn);

            for (INS ins = RTN_InsHead(rtn); INS_Valid(ins);
                                                ins = INS_Next(ins))
            {
                count++;
            }

            // to preserve space, release data associated with the RTN
            RTN_Close(rtn);

            fprintf(trace, "%s has %d insts\n",
                                        RTN_Name(rtn).c_str(), count);
        }
    }
}
```

```
// Pin calls this function every time a new img is unloaded
// You can't instrument an image that is about to be unloaded
VOID ImageUnload(IMG img, VOID *v)
{
    fprintf(trace, "Unloading %s\n", IMG_Name(img).c_str());
}

// This function is called when the application exits
// It prints the name and count for each procedure
VOID Fini(INT32 code, VOID *v)
{
    fclose(trace);
}

// argc, argv are the entire command line, including pin -t <toolname> -- ...
int main(int argc, char * argv[])
{
    trace = fopen("imageload.out", "w");

    // Initialize symbol processing
    PIN_InitSymbols();

    // Initialize pin
    PIN_Init(argc, argv);

    // Register ImageLoad to be called when an image is loaded
    IMG_AddInstrumentFunction(ImageLoad, 0);

    // Register ImageUnload to be called when an image is unloaded
    IMG_AddUnloadFunction(ImageUnload, 0);

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

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

    return 0;
}
```

Replacing a Routine in Probe Mode

Probe mode is a method of using Pin to insert probes at the start of specified routines. In probe mode, the application and the replacement routine are run natively. This improves performance, but it puts more responsibility on the tool writer. Probes can only be placed on RTN boundaries.

A probe is a jump instruction that is placed at the start of the specified routine. The probe redirects the flow of control to the replacement function. Before the probe is inserted, the first few instructions of the specified routine are relocated. It is not uncommon for the replacement function to call the replaced routine. Pin provides the relocated address to facilitate this.

In the following Pintool, the tool replaces an original library function (`malloc`) with a custom function defined in the Pintool (`NewMalloc`) using the Probe Mode. The custom function is called instead of the program's original function, and it is useful for very fast function-level profiling (e.g. tracing thread synchronization primitives), and the original function is invoked thereafter using the function pointer handed over upon replacing the application's original function.

NOTE: The command line requires the "`-probe`" mode argument.

```
$ ../Bin/pin -probe -t ./newmalloc.so -- /bin/ls

Replacing malloc in /lib/ld-linux.so.2
Org.  entry to replaced function has been moved to 0xb6826158
NewMalloc (b6826158, 589)
NewMalloc (b6826158, 87)
NewMalloc (b6826158, 12)
NewMalloc (b6826158, 960)
...

$
```

```
#include "pin.H"
#include <iostream>
using namespace std;

typedef VOID * (*FP_MALLOC)(size_t);

// This is the replacement routine.
VOID * NewMalloc(FP_MALLOC orgFuncptr, UINT32 arg0)
{
    // Normally one would do something more interesting with this data.
    cout << "NewMalloc ("
         << hex << ADDRINT (orgFuncptr) << ", "
         << dec << arg0 << ")"
         << endl << flush;

    // Call the relocated entry point of the original (replaced) routine.
    VOID * v = orgFuncptr(arg0);

    return v;
}

// Pin calls this function every time a new img is loaded.
// It is best to do probe replacement when the image is loaded,
// because only one thread knows about the image at this time.
VOID ImageLoad(IMG img, VOID *v)
{
    // See if malloc() is present in the image.  If so, replace it.
    RTN rtn = RTN_FindByName(img, "malloc");

    if (RTN_Valid(rtn))
    {
        cout << "Replacing malloc in " << IMG_Name(img) << endl;

        // Define a function prototype that describes the application routine
        // that will be replaced.
        PROTO proto_malloc = PROTO_Allocate(PIN_PARG(void *),
                                            CALLINGSTD_DEFAULT,
                                            "malloc",
                                            PIN_PARG(int),
                                            PIN_PARG_END());

        // Replace the application routine with the replacement function.
        // Additional arguments have been added to the replacement routine.
        // The return value and the argument passed into the replacement
        // function with IARG_ORIG_FUNCPTR are the same.
        AFUNPTR origptr = RTN_ReplaceSignatureProbed(rtn, AFUNPTR(NewMalloc),
                              IARG_PROTOTYPE, proto_malloc,
                              IARG_ORIG_FUNCPTR,
                              IARG_FUNCARG_ENTRYPOINT_VALUE, 0,
                              IARG_END);

        cout << "Org. entry to replaced function has been moved to 0x";
        cout << hex << (ADDRINT) origptr << dec << endl;
```

```
        // Free the function prototype.
        PROTO_Free(proto_malloc);
    }
}

// Initialize and start Pin in Probe mode.  "-probe" must be used on
// the command line.
int main(INT32 argc, CHAR *argv[])
{
    // Initialize symbol processing
    PIN_InitSymbols();

    // Initialize pin
    PIN_Init(argc, argv);

    // Register ImageLoad to be called when an image is loaded
    IMG_AddInstrumentFunction(ImageLoad, 0);

    // Start the program in probe mode, never returns
    PIN_StartProgramProbed();

    return 0;
}
```

Extending Function Signatures

The "`NewMalloc`" function call does not need to have the same function signature as the original function call. Let us extend the prototype of the function dynamically! This is extremely useful for passing various Pin IARG's, as well as your own custom arguments (such as a specific memory pool from which to allocate based on the call-site).

In the following examples we will trace the callers of the application's `malloc` function by requesting Pin for the call-site return address and use that address to lookup the name of the caller function. The expected output of the tool is show below:

```
$ ../Bin/pin -probe -t ./newmalloc.so -- /bin/ls

Replacing malloc in /lib/ld-linux.so.2
Org.  entry to replaced function has been moved to 0xb6826158
NewMalloc (b6826158, 589, calloc)
NewMalloc (b6826158, 87, _dl_important_hwcaps)
NewMalloc (b6826158, 12, _dl_init_paths)
NewMalloc (b6826158, 960, _dl_init_paths)
NewMalloc (b6826158, 8, _dl_init_paths)
NewMalloc (b6826158, 64, fillin_rpath)
NewMalloc (b6826158, 20, _dl_map_object)
...

$
```

Follow the below instructions to identify the arguments of the `malloc` function, as well as print the name of the function calling `malloc`:

Step 1: Extend the IARG of `RTN_ReplaceSignatureProbed` by asking Pin to pass the return address of the caller function. This is achieved using `IARG_RETURN_IP`.

Step 2: Extend the signature of our `NewMalloc` function to accept the return address by adding "`ADDRINT returnIp`" to the function signature.

Step 3: Print the name of the caller by looking up the caller's function name using `RTN_FindNameByAddress` with `returnIp` as its argument.

```
#include "pin.H"
#include <iostream>
using namespace std;

typedef VOID * (*FP_MALLOC)(size_t);

// This is the replacement routine.
VOID * NewMalloc(FP_MALLOC orgFuncptr, UINT32 arg0, ADDRINT returnIp)
{
    // Normally one would do something more interesting with this data.
    cout << "NewMalloc ("
         << hex << ADDRINT (orgFuncptr) << ", "
         << dec << arg0 << ", "
         << RTN_FindNameByAddress(returnIp) << ")"
         << endl << flush;

    // Call the relocated entry point of the original (replaced) routine.
    VOID * v = orgFuncptr(arg0);

    return v;
}

// Pin calls this function every time a new img is loaded.
// It is best to do probe replacement when the image is loaded,
// because only one thread knows about the image at this time.
VOID ImageLoad(IMG img, VOID *v)
{
    // See if malloc() is present in the image.  If so, replace it.
    RTN rtn = RTN_FindByName(img, "malloc");

    if (RTN_Valid(rtn))
    {
        cout << "Replacing malloc in " << IMG_Name(img) << endl;

        // Define a function prototype that describes the application routine
        // that will be replaced.
        PROTO proto_malloc = PROTO_Allocate(PIN_PARG(void *),
                                            CALLINGSTD_DEFAULT,
                                            "malloc",
```

```
                                                PIN_PARG(int),
                                                PIN_PARG_END());

        // Replace the application routine with the replacement function.
        // Additional arguments have been added to the replacement routine.
        // The return value and the argument passed into the replacement
        // function with IARG_ORIG_FUNCPTR are the same.
        AFUNPTR origptr = RTN_ReplaceSignatureProbed(rtn, AFUNPTR(NewMalloc),
                                IARG_PROTOTYPE, proto_malloc,
                                IARG_ORIG_FUNCPTR,
                                IARG_FUNCARG_ENTRYPOINT_VALUE, 0,
                                IARG_RETURN_IP,
                                IARG_END);

        cout << "Org. entry to replaced function has been moved to 0x";
        cout << hex << (ADDRINT) origptr << dec << endl;

        // Free the function prototype.
        PROTO_Free(proto_malloc);
    }
}

// Initialize and start Pin in Probe mode.  "-probe" must be used on
// the command line.
int main(INT32 argc, CHAR *argv[])
{
    // Initialize symbol processing
    PIN_InitSymbols();

    // Initialize pin
    PIN_Init(argc, argv);

    // Register ImageLoad to be called when an image is loaded
    IMG_AddInstrumentFunction(ImageLoad, 0);

    // Start the program in probe mode, never returns
    PIN_StartProgramProbed();

    return 0;
}
```

Build Your Own Pin Tool -- *strace* Using Pin

> strace is a tool for tracing system calls and signals. It intercepts and records the system calls made by a running process, and can print a record of each system call, its arguments, and its return value. strace is extremely useful as a diagnostic, instructional, and debugging tool. System administrators, diagnosticians and trouble-shooters will find it invaluable for solving problems with programs for which the source is not readily available since they do not need to be recompiled in order to trace them. Students, hackers and the overly-curious will find that a great deal can be learned about a system and its system calls by tracing even ordinary programs. And programmers will find that since system calls and signals are events that happen at the user/kernel interface, a close examination of this boundary is very useful for bug isolation, sanity checking and attempting to capture race conditions.

Running strace generates the following output:

```
$ strace /bin/echo

execve("/bin/echo", ["echo"], [/* 31 vars */]) = 0
uname({sys="Linux", node="ubuntu-desktop", ...}) = 0
brk(0)                                  = 0x804d000
…
open("/etc/ld.so.cache", O_RDONLY)      = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=42307, ...}) = 0
old_mmap(NULL, 42307, PROT_READ, MAP_PRIVATE, 3, 0) = 0xb7efa000
…

$
```

The goal is to modify one of your prior tools to mimic the strace program functionality of tracing system calls. The following steps are a rough guide through the process:

Step 1: strace is concerned only with system calls, so your instrumentation function must intercept only system calls. Wrap your instruction instrumentation call (i.e. `INS_InsertCall`) around with the `INS_IsSyscall` inspection check, which returns true only for system call instructions:

```
if (INS_IsSyscall(ins))
{
      INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR) print, IARG_END);
}
```

Step 2: Tracing system calls is insightful only if we know the system call being executed by the program. This information can be requested from Pin using the `IARG_SYSCALL_NUMBER` instrumentation argument. `IARG_SYSCALL_NUMBER` passes the number of a system call to the analysis function. Modify your instrumentation, as well as the analysis function to intercept and

print the system call number as follows:

```
VOID print(INT32 num)
{
      printf("%d \n", num);
}

VOID Instruction(INS ins, VOID *v)
{
      if (INS_IsSyscall(ins))
      {
            INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR) print,
                                          IARG_SYSCALL_NUMBER, IARG_END);
      }
}
```

Building and running the Pin Tool should now produce an output that resembles the following:

```
 122
 45
 90
 33
 90
 33
 5
 …
```

Step 3: Next, we are interested in dissecting the arguments passed by the application to the system call interface. Using `IARG_SYSCALL_VALUE, n` a Pin Tool can intercept the n^{th} system call argument as shown below:

```
VOID print(INT32 num, VOID * arg0,
           VOID * arg1, VOID * arg2, VOID * arg3, VOID * arg4, VOID * arg5)
{
      printf("%d(%p, %p, %p, %p, %p, %p) \n",
                                    num, arg0, arg1, arg2, arg3, arg4, arg5);
}

VOID Instruction(INS ins, VOID *v)
{
      if (INS_IsSyscall(ins))
      {
            INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR) print,
                                    IARG_SYSCALL_NUMBER,
                                    IARG_SYSARG_VALUE, 0,
                                    IARG_SYSARG_VALUE, 1,
                                    IARG_SYSARG_VALUE, 2,
                                    IARG_SYSARG_VALUE, 3,
                                    IARG_SYSARG_VALUE, 4,
                                    IARG_SYSARG_VALUE, 5,
                                    IARG_END);
```

```
        }
}
```

Your Pin Tool's output should now start taking the form of the output generated by strace:

```
122(0xbf89ffbc, (nil), 0xb7fa3778, (nil), 0xb7f8e000, 0xbf8a0198)
45((nil), 0xb7fa3778, 0xffffe400, 0x2060f, 0xbf8a0045, 0xbf89ff84)
90(0xbf89fd3c, 0x24d, 0xb7fa3778, 0xb7fa3f08, 0x1000, 0xbf89fd60)
33(0xb7fa1e7b, (nil), 0xb7fa3778, 0xb6784020, 0x804c280, 0xbf89fd84)
90(0xbf89fd4c, 0x1838, 0xb7fa3778, 0xb6784320, 0x2000, 0xbf89fd70)
33(0xb7f9f54d, 0x4, 0xb7fa3778, 0xfffffffe0, 0xb7f9f54d, 0xbf89ff84)
5(0xb7fa1e8e, (nil), (nil), 0xb67841dc, 0xb7fa3e70, 0xbf89f8c8)
…
```

Step 4: There is still one remaining step to complete the Pin-based strace tool – we must find the return value of a system call. The return value of a system call can be requested from Pin using `IARG_SYSRET_VALUE`. Since the return value is only available after a system call instruction finishes execution, a new instrumentation call is required that invokes a different analysis routine to print the return value. Notification of system call completion can be got using `IPOINT_AFTER` instrumentation. To achieve this required functionality, add the following code:

```
VOID printafter(VOID * ret)
{
        printf("returns: %p \n", ret);
}


VOID Instruction(INS ins, VOID *v)
{
        if (INS_IsSyscall(ins))
        {
                …
                INS_InsertCall(ins, IPOINT_AFTER, AFUNPTR(printafter),
                                                  IARG_SYSRET_VALUE,
                                                  IARG_END);
        }
}
```

While program output varies from one system to another, the general output form should resemble the following:

```
 33(0xb7fa654d, 0x4, 0xb7faa778, 0xfffffffe0, 0xb7fa654d, 0xbffa8e14)
returns: 0xffffffff
5(0xb7fa8e8e, (nil), (nil), 0xb678c1dc, 0xb7faae70, 0xbffa8758)
returns: 0x3
...
```

You have now successfully built your own strace Pin Tool (in < 50 lines)!

Conclusion

In summarizing today's hands-on experience, we have improved an instruction counting tool to perform instruction and memory address tracing. We have also covered how to attach a debugger to help debug a Pintool. Lastly, we took a peek into how Pin can be used to alter application code dynamically. Overall, Pin exposes a transparent API that is easy to use, flexible, and (mostly) independent of the underlying platform, thereby allow users to focus on the task at hand rather than expend energy dealing with ISA and ABI idiosyncrasies.

Many of the examples covered in this hands-on session are directly derived from the examples provided in the Pin kit. The best starting point for writing your own tool is to modify an existing Pintool slightly to suit your needs.

Congratulations -- you are now a Pinhead!