Pin: Intel’s Dynamic Binary Instrumentation Engine

Pin Tutorial

Intel Corporation

Written By: Tevi Devor

Presented By: Sion Berkowits

CGO 2013
Legal Disclaimer

ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

Performance tests and ratings are measured using specific computer systems and/or components and reflect the approximate performance of Intel products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance. Buyers should consult other sources of information to evaluate the performance of systems or components they are considering purchasing. For more information on performance tests and on the performance of Intel products, reference www.intel.com/software/products.

Intel and the Intel logo are trademarks of Intel Corporation in the U.S. and other countries.

*Other names and brands may be claimed as the property of others.

Copyright © 2013. Intel Corporation.
Intel compilers, associated libraries and associated development tools may include or utilize options that optimize for instruction sets that are available in both Intel and non-Intel microprocessors (for example SIMD instruction sets), but do not optimize equally for non-Intel microprocessors. In addition, certain compiler options for Intel compilers, including some that are not specific to Intel microarchitecture, are reserved for Intel microprocessors. For a detailed description of Intel compiler options, including the instruction sets and specific microprocessors they implicate, please refer to the “Intel Compiler User and Reference Guides” under “Compiler Options.” Many library routines that are part of Intel compiler products are more highly optimized for Intel microprocessors than for other microprocessors. While the compilers and libraries in Intel compiler products offer optimizations for both Intel and Intel-compatible microprocessors, depending on the options you select, your code and other factors, you likely will get extra performance on Intel microprocessors.

Intel compilers, associated libraries and associated development tools may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include Intel® Streaming SIMD Extensions 2 (Intel® SSE2), Intel® Streaming SIMD Extensions 3 (Intel® SSE3), and Supplemental Streaming SIMD Extensions 3 (Intel SSSE3) instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors.

While Intel believes our compilers and libraries are excellent choices to assist in obtaining the best performance on Intel and non-Intel microprocessors, Intel recommends that you evaluate other compilers and libraries to determine which best meet your requirements. We hope to win your business by striving to offer the best performance of any compiler or library; please let us know if you find we do not.

Notice revision #20110307
Agenda

• **Part 1**: Introduction to Pin

• **Part 2**: Topics in Pin API

• **Part 3**: Performance – Optimizing your Pin tool

• **Part 4**: Advanced Pin
Part 1
Introduction to Pin
Instrumentation in a nutshell

• A technique that inserts code into a program to collect run-time information

```assembly
movzx ecx, [rax+0x2]
call 0x77ef7870
cmp rax, rdx
jz 0x77f1eac9
```
Instrumentation types

• Different usages
  – Program analysis: performance profiling, error detection, capture & replay
  – Architectural study: processor and cache simulation, trace collection
  – Binary translation: Modify program behavior, emulate unsupported instructions

• Different types
  – Source code instrumentation
  – Static binary instrumentation
  – Dynamic binary instrumentation
Dynamic binary instrumentation

• Instrument binary code right before it runs
  – a.k.a. Just in time, or JIT

• Benefits
  – No need to recompile or re-link
  – Discover code at runtime
  – Handle dynamically generated code
  – Attach to running processes
Pin

- Dynamic binary instrumentation framework
- Developed at Intel

- What does “Pin” stand for?
  - Pin Is Not an acronym
  - Pin is based on the IPF post link optimizer iSpike
  - Pin is a small Spike
    - Spike is EOL
      [link](http://www.cgo.org/cgo2004/papers/01_82_luk_ck.pdf)
Advantages of Pin Instrumentation

• **Programmable Instrumentation:**
  – Write your own instrumentation tools, called PinTools
    – PinTools can be written in C, C++, assembly
    – APIs are designed to maximize ease of use
      – abstract away the underlying instruction set idiosyncrasies

• **Multiplatform:**
  – OS’s: Windows, Linux, OSX, Android
  – Architectures: IA-32, Intel64

• **Robust:**
  – Instruments real-life applications: Database, web browsers, ...
  – Instruments multithreaded applications
  – Supports signals and exceptions, self modifying code...

• **Efficient:**
  – Applies compiler optimizations on instrumentation code

*Pin can be used to instrument all the user level code in an application*
PinTool Capabilities

• Replace application functions with your own
  – Call the original function from within your function

• Fully examine any application instruction, insert a call to your instrumenting function to be executed whenever that instruction executes
  – Pass parameters to your instrumenting function from a large set of supported parameters
    – Register values (including IP), also by reference (for modification)
    – Memory addresses read/written by the instruction
    – Full registers context
    – ...

• Track function calls, including syscalls
  – Examine/change arguments

• Track application threads
• Intercept signals
• Instrument a process tree
• Many other capabilities...
Usage of Pin at Intel

• Profiling and analysis products
  – Intel® Parallel Studio XE
    – Intel® VTune™ Amplifier XE (performance analysis)
      – Locks and waits analysis
      – Concurrency analysis
    – Intel® Inspector XE (correctness analysis)
      – Threading error detection (data race and deadlock)
      – Memory error detection

• Architectural research and enabling
  – Emulating new instructions (Intel SDE)
  – Trace generation
  – Branch prediction and cache modeling

• Others
  – PinPlay, PinPoints (go to HPCS tutorial later today)
Pin Usage Outside Intel

• Popular and well supported
  – 30,000+ downloads, 700+ citations

• Free Download
  – www.pintool.org
  – Includes: Detailed user manual, source code for 100s of Pin tools

• Pin User Group (PinHeads)
  – http://tech.groups.yahoo.com/group/pinheads/
  – Pin users and Pin developers answer questions
Example Pin invocation

• Application:
  gzip.exe input.txt

• PinTool: inscount.dll
  – Count application instructions executed, print count at end

• Invocation:

  > pin.exe -t inscount.dll -- gzip.exe input.txt
pin.exe -t inscount.dll -- gzip.exe input.txt
pin.exe –t inscount.dll -- gzip.exe input.txt

Launcher Process

PIN.EXE

Launcher

Application Process

inscount.dll
PIN.LIB

PINVM.DLL

System Call Dispatcher
Event Dispatcher
Thread Dispatcher

Code Cache

NTDLL.DLL

Windows kernel

Boot Routine + Data: firstAppIp, "Inscount.dll"

Application Code and Data

app Ip of Trace's target

First app IP
All code in this presentation is covered by the following:

- /*BEGIN_LEGAL
- Intel Open Source License

- Copyright (c) 2002-2013 Intel Corporation. All rights reserved.

- Redistribution and use in source and binary forms, with or without
  modification, are permitted provided that the following conditions are
  met:

  Redistributions of source code must retain the above copyright notice,
  this list of conditions and the following disclaimer. Redistributions
  in binary form must reproduce the above copyright notice, this list of
  conditions and the following disclaimer in the documentation and/or
  other materials provided with the distribution. Neither the name of
  the Intel Corporation nor the names of its contributors may be used to
  endorse or promote products derived from this software without
  specific prior written permission.

- THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
  `AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
  LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
  A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE INTEL OR
  ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
  SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
  LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
  DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
  THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
  (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
  OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

- END_LEGAL */
**Instruction Counting Tool (inscount.dll)**

```c
#include "pin.h"

UINT64 icount = 0;

void docount() { icount++; }

void Instruction(INS ins, void *v)
{
    INS_InsertCall(ins, IPOINT_BEFORE,
                   (AFUNPTR)docount, IARG_END);
}

void Fini(INT32 code, void *v)
{
    std::cerr << "Count " << icount << endl;
}

int main(int argc, char * argv[])
{
    PIN_Init(argc, argv);
    INS_AddInstrumentFunction(Instruction, 0);
    PIN_AddFiniFunction(Fini, 0);
    PIN_StartProgram(); // Never returns
    return 0;
}
```

*Execution time routine*

*Jitting time routine: Pin CallBack*

```c
mov    0x1, %edi
jle    <L1>

inc icount
sub $0xff, %edx
cmp %esi, %edx
save eflags
inc icount
restore eflags
jle    <L1>
inc icount
mov 0x1, %edi
```
Instrumentation vs. Analysis

• **Instrumentation routines** define where instrumentation is inserted
  – e.g., before instruction
  ✤ Occurs *when* an instruction is being jitted

• **Analysis routines** define what to do when instrumentation is activated
  – e.g., increment counter
  ✤ Occurs *every time* an instruction is executed
• Trace: A sequence of continuous instructions, with one entry point
• BBL: has one entry point and ends at first control transfer instruction
```c
#include "pin.H"

UINT64 icount = 0;

void PIN_FAST_ANALYSIS_CALL docount(INT32 c) { icount += c; }

void Trace(TRACE trace, void *v){
  // Pin Callback
  for(BBL bbl = TRACE_BblHead(trace);
      BBL_Valid(bbl);
      bbl = BBL_Next(bbl))
    BBL_InsertCall(bbl, IPOINT_ANYWHERE,
                   (AFUNPTR)docount, IARG_FAST_ANALYSIS_CALL,
                   IARG_UINT32, BBL_NumIns(bbl),
                   IARG_END);
}

void Fini(INT32 code, void *v) {
  // Pin Callback
  fprintf(stderr, "Count %lld\n", icount);
}

int main(int argc, char * argv[])
{
  PIN_Init(argc, argv);
  TRACE_AddInstrumentFunction(Trace, 0);
  PIN_AddFiniFunction(Fini, 0);
  PIN_StartProgram();
  return 0;
}
```
APP IP
0x77ec4600 cmp rax, rdx
0x77ec4603 jz 0x77f1eac9
0x77ec4609 movzx ecx, [rax+0x2]
0x77ec460d call 0x77ef7870

```
0x001de0000 mov r14, 0xc5267d40 //inscount2.docount
0x001de000a add [r14], 0x2    //inscount2.docount
0x001de0015 0x77ec4600 cmp rax, rdx
0x001de0018 jz 0x1deffa0  L1 //patched in future
0x001de001e mov r14, 0xc5267d40 //inscount2.docount
0x001de0028 mov [r15+0x60], rax
0x001de002c mov [r15+0x60], rax
0x001de002e mov [r15+0x60], rax
0x001de0033 cmp rax, rdx
0x001de0036 add [r14], 0x2    //inscount2.docount
0x001de003c movzx ecx, [rax+0x2] //ecx alloced to edi
0x001de0040 movzx ecx, [rax+0x2]
0x001de0044 push 0x77ec4612    //push retaddr
0x001de0048 cmp rax, rdx
0x001de004b movzx ecx, [rax+0x2] //ecx alloced to edi
0x001de0051 mov rax, [r15+0x60] //inscount2.docount
0x001de0055 add [r14], 0x2    //inscount2.docount
0x001de005b movzx ecx, [rax+0x2] //ecx alloced to edi
0x001de0061 movzx ecx, [rax+0x2]
0x001de0065 push 0x77ec4612    //push retaddr
0x001de0069 cmp rax, rdx
0x001de006c movzx ecx, [rax+0x2] //ecx alloced to edi
0x001de0070 movzx ecx, [rax+0x2]
0x001de0074 push 0x77ec4612    //push retaddr
0x001de0078 cmp rax, rdx
0x001de007b movzx ecx, [rax+0x2] //ecx alloced to edi
0x001de007f movzx ecx, [rax+0x2]
0x001de0083 push 0x77ec4612    //push retaddr
0x001de0087 cmp rax, rdx
0x001de008a movzx ecx, [rax+0x2] //ecx alloced to edi
0x001de008e movzx ecx, [rax+0x2]
0x001de0092 push 0x77ec4612    //push retaddr
0x001de0096 cmp rax, rdx
0x001de0099 movzx ecx, [rax+0x2] //ecx alloced to edi
0x001de009d movzx ecx, [rax+0x2]
0x001de00a1 push 0x77ec4612    //push retaddr
0x001de00a5 cmp rax, rdx
0x001de00a8 movzx ecx, [rax+0x2] //ecx alloced to edi
0x001de00aC movzx ecx, [rax+0x2]
0x001de00B0 push 0x77ec4612    //push retaddr
0x001de00B4 cmp rax, rdx
0x001de00B7 movzx ecx, [rax+0x2] //ecx alloced to edi
0x001de00BB movzx ecx, [rax+0x2]
0x001de00BD push 0x77ec4612    //push retaddr
0x001de00BE cmp rax, rdx
0x001de00C1 movzx ecx, [rax+0x2] //ecx alloced to edi
0x001de00C5 movzx ecx, [rax+0x2]
0x001de00C9 push 0x77ec4612    //push retaddr
0x001de00CB cmp rax, rdx
0x001de00CE movzx ecx, [rax+0x2] //ecx alloced to edi
0x001de00D0 movzx ecx, [rax+0x2]
0x001de00D3 push 0x77ec4612    //push retaddr
```

L2:
```
0x001deffd0 mov [r15+0x40], rsp // save app rsp
0x001deffd4 mov rsp, [r15+0x2d0] // switch to pin stack
0x001deffdb call [0x2f000000]  //call VmEnter
// data used by VmEnter – pointed to by return-address of call
0x001deffe8_svc(VMSVC_XFER)
0x001deff0_sct(0x00065fb60) // current register mapping
0x001defff8_iaddr(0x077ef7870) // app target IP of
// call at 0x77ec460d
```

L1:
```
0x001deffa0 mov [r15+0x40], rsp // save app rsp
0x001deffa4 mov rsp, [r15+0x2d0] // switch to pin stack
0x001deffab call [0x2f000000]  //call VmEnter
// data used by VmEnter – pointed to by return-address of call
0x001deff8_svc(VMSVC_XFER)
0x001deffc0_sct(0x00065f998) // current register mapping
0x001deff8_iaddr(0x077f1eac9) // app target IP of jz at 0x77ec4603
```
Multi-Threading

• Pin supports multi-threading

  - Application threads execute jitted code including instrumentation code (inlined and not inlined), without any serialization introduced by Pin
    - Instrumentation code can use Pin and/or OS synchronization constructs to introduce serialization if needed.
    - Pin provides APIs for thread local storage.
  - Pin callbacks are serialized
  - Jitting is serialized
    - Only one application thread can be jitting code at any time
#include "pin.H"
INT32 numThreads = 0;
const INT32 MaxNumThreads = 10000;
struct THREAD_DATA
{
    UINT64 _count;
    UINT8 _pad[56]; /* guess why? */ }icount[MaxNumThreads];

// Analysis routine
VOID PIN_FAST_ANALYSIS_CALL docount(ADDRINT c, THREADID tid) { icount[tid]._count += c;}
// Pin Callback
VOID ThreadStart(THREADID threadid, CONTEXT *ctxt, INT32 flags, VOID *v){numThreads++;}

VOID Trace(TRACE trace, VOID *v) { // Jitting time routine: Pin Callback
    for (BBL bbl = TRACE_BblHead(trace); BBL_Valid(bbl); bbl = BBL_Next(bbl))
        BBL_InsertCall(bbl, IPOINT_ANYWHERE, (AFUNPTR)dcount, IARG_FAST_ANALYSIS_CALL,
            IARG_UINT32, BBL_NumIns(bbl), IARG_THREAD_ID, IARG_END);
}

VOID Fini(INT32 code, VOID *v){
    for (INT32 t=0; t<numThreads; t++)
        printf("InsCount[of thread#%d]= %d\n",t,icount[t]._count);
}

int main(int argc, char * argv[])
{
    PIN_Init(argc, argv);
    for (INT32 t=0; t<MaxNumThreads; t++) {icount[t]._count = 0;}
    PIN_AddThreadStartFunction(ThreadStart, 0);
    TRACE_AddInstrumentFunction(Trace, 0);
    PIN_AddFiniFunction(Fini, 0);
    PIN_StartProgram(); return 0;  }
A couple more examples..
#include "pin.h"
#include <map>
std::map<ADDRINT, std::string> disAssemblyMap;

VOID ReadsMem (ADDRINT applicationIp, ADDRINT memoryAddressRead, UINT32 memoryReadSize) {
    printf("0x%x %s  reads %d bytes of memory at 0x%x\n", applicationIp, disAssemblyMap[applicationIp].c_str(), memoryReadSize, memoryAddressRead);
}

VOID Instruction(INS ins, void * v) {
    // Jitting time routine
    if (INS_IsMemoryRead(ins)) {
        disAssemblyMap[INS_Address(ins)] = INS_Disassemble(ins);
        INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR) ReadsMem,
                       IARG_INST_PTR, // application IP
                       IARG_MEMORYREAD_EA,
                       IARG_MEMORYREAD_SIZE,
                       IARG_END);
    }
}

int main(int argc, char * argv[]) {
    PIN_Init(argc, argv);
    INS_AddInstrumentFunction(Instruction, 0);
    PIN_StartProgram();
}
**SDE**

- **SDE**: A fast functional simulator for applications with new instructions
  - New instructions have been defined
  - Compiler generates code with new instructions
  - What can be used to run the apps with the new instructions?
    - Use PinTool that emulates new instructions.

  - `vmovdqu ymm?, mem256        vmovdqu mem256, ymm?`
    - 16 new 256 bit ymm registers
    - Read/Write ymm register from/to memory.
#include "pin.H"

VOID EmVmovdquMem2Reg(unsigned int ymmDstRegNum, ADDRINT * ymmMemSrcPtr) {
    PIN_SafeCopy(ymmRegs[ymmDstRegNum], ymmMemSrcPtr, 32);
}

VOID EmVmovdquReg2Mem(int ymmSrcRegNum, ADDRINT * ymmMemDstPtr) {
    PIN_SafeCopy(ymmMemDstPtr, ymmRegs[ymmRegNum], 32);
}

VOID Instruction(INS ins, VOID *v) {
    switch (INS_Opcode(ins)) {
        :::::
        case XED_ICLASS_VMOVDQU:
            if (INS_IsMemoryRead(ins)) // vmovdqu ymm? <= mem256
                INS_InsertCall(ins, IPOINT_BEFORE,
                               (AFUNPTR)EmVmovdquMem2Reg,
                               IARG_UINT32, REG(INS_OperandReg(ins, 0)) - REG_YMM0,
                               IARG_MEMORYREAD_EA,
                               IARG_END);
            else if (INS_IsMemoryWrite(ins)) // vmovdqu mem256 <= ymm?
                INS_InsertCall(ins, IPOINT_BEFORE,
                               (AFUNPTR)EmVmovdquReg2Mem,
                               IARG_UINT32, REG(INS_OperandReg(ins, 1)) - REG_YMM0,
                               IARG_MEMORYWRITE_EA,
                               IARG_END);
            INS_DeleteIns(ins); // Processor does NOT execute this instruction
            break;
        }
} }

int main(int argc, CHAR *argv[]) {
    PIN_Init(argc,argv);
    INS_AddInstrumentFunction(Instruction, 0);
    PIN_StartProgram();
}
Symbols
Probe-mode
The CONTEXT structure
Multi-threading
Instrumenting a process tree

Part 2
Topics in Pin API
Symbols
Symbols

• *PIN_InitSymbols()*
  - Pin will use whatever symbol information is available
    - Debug info in the app
    - Pdb files
    - Export Tables
    - On Windows uses dbghelp
  - See *PIN_InitSymbolsAlt()* for more control over which symbols will be used

• Use symbols to instrument/wrap/replace specific functions

• Access application debug information from a Pin tool
  - Use API function *PIN_GetSourceLocation()*
Instrument malloc and free

int main(int argc, char *argv[])
{
    // Initialize pin symbol manager
    PIN_InitSymbols();
    // See also PIN_InitSymbolsAlt() for more control over which symbols are read
    PIN_Init(argc, argv);

    // Register the function ImageLoad to be called each time an image is loaded in the process
    // This includes the process itself and all shared libraries it loads (implicitly or explicitly)
    IMG_AddInstrumentFunction(ImageLoad, 0);

    // Never returns
    PIN_StartProgram();
}
VOID ImageLoad(IMG img, VOID *v) // Pin Callback.
{
    // Instrument the malloc() and free() functions. Print the input argument
    // of each malloc() or free(), and the return value of malloc().

    RTN mallocRtn = RTN_FindByName(img, "_malloc"); // Find the malloc() function.
    if (RTN_Valid(mallocRtn))
    {
        RTN_Open(mallocRtn);

        // Instrument malloc() to print the input argument value and the return value.
        RTN_InsertCall(mallocRtn, IPOINT_BEFORE, (AFUNPTR)MallocBefore,
                        IARG_FUNCARG_ENTRYPOINT_VALUE, 0,
                        IARG_END);
        RTN_InsertCall(mallocRtn, IPOINT_AFTER, (AFUNPTR)MallocAfter,
                        IARG_FUNCRET_EXITPOINT_VALUE, IARG_END);
        RTN_Close(mallocRtn);
    }

    RTN freeRtn = RTN_FindByName(img, "_free"); // Find the free() function.
    if (RTN_Valid(freeRtn))
    {
        RTN_Open(freeRtn);
        // Instrument free() to print the input argument value.
        RTN_InsertCall(freeRtn, IPOINT_BEFORE, (AFUNPTR)FreeBefore,
                        IARG_FUNCARG_ENTRYPOINT_VALUE, 0,
                        IARG_END);
        RTN_Close(freeRtn);
    }
}
Alternative: Handling name-mangling and multiple symbols at same address

VOID Image(IMG img, VOID *v)
{
    // Walk through the symbols in the symbol table.
    for (SYM sym = IMG_RegsymHead(img); SYM_Valid(sym); sym = SYM_Next(sym))
    {
        string undFuncName = PIN_UndecorateSymbolName(SYM_Name(sym), UNDECORATION_NAME_ONLY);
        if (undFuncName == "malloc") // Find the malloc function.
        {
            RTN mallocRtn = RTN_FindByAddress(IMG_LowAddress(img) + SYM_Value(sym));

            if (RTN_Valid(mallocRtn))
            {
                RTN_Open(mallocRtn);

                // Instrument to print the input argument value and the return value.
                RTN_InsertCall(mallocRtn, IPOINT_BEFORE, (AFUNPTR)MallocBefore,
                               IARG_FUNCARG_ENTRYPOINT_VALUE, 0,
                               IARG_END);
                RTN_InsertCall(mallocRtn, IPOINT_AFTER, (AFUNPTR)MallocAfter,
                               IARG_FUNCRET_EXITPOINT_VALUE,
                               IARG_END);

                RTN_Close(mallocRtn);
            }
        }
    }
}
VOID Instruction(INS ins, VOID *v) // INS_AddInstrumentFunction(Instruction, 0);
{
    UINT32 numMemOperands = INS_MemoryOperandCount(ins);

    // Iterate over each memory operand of the instruction.
    for (UINT32 memOp = 0; memOp < numMemOperands; memOp++)
    {
        if (INS_MemoryOperandIsWritten(ins, memOp))
        {
            // Insert instrumentation code to catch a memory overwrite
            INS_InsertIfCall (ins, IPOINT_BEFORE, AFUNPTR(AnalyzeMemWrite),
                              IARG_FAST_ANALYSIS_CALL, IARG_MEMORYOP_EA, memOp,
                              IARG_MEMORYWRITE_SIZE, IARG_END);

            INS_InsertThenCall (ins, IPOINT_BEFORE, AFUNPTR(MemoryOverWriteAt),
                                IARG_FAST_ANALYSIS_CALL, IARG_INST_PTR,
                                IARG_MEMORYOP_EA, memop,
                                IARG_MEMORYWRITE_SIZE, IARG_END);
        }
    }
}
Accessing Application Debug Info from a Pin Tool: Catch a Memory Overwrite

KNOB<ADDRINT> KnobMemAddrBeingOverwritten(KNOB_MODE_WRITEONCE, "pintool",
"mem_overwrite_addr", "256", "overwritten memaddr");

static ADDRINT PIN_FAST_ANALYSIS_CALL
AnalyzeMemWrite ( // Pin will inline this function, it is the IF part
ADDRINT memWriteAddr, UINT32 numBytesWritten)
{
    // return 1 if this memory write overwrites the address specified by
    // KnobMemAddrBeingOverwritten
    return (memWriteAddr <= KnobMemAddrBeingOverwritten &&
            (memWriteAddr + numBytesWritten) > KnobMemAddrBeingOverwritten);
}

static VOID PIN_FAST_ANALYSIS_CALL
MemoryOverWriteAt ( // Pin will NOT inline this function, it is the THEN part
ADDRINT appIP, ADDRINT memWriteAddr, UINT32 numBytesWritten)
{
    INT32 column, lineNum;
    string fileName;

    PIN_GetSourceLocation (appIP, &column, &line, &fileName);

    printf ("overwrite of %p from instruction at %p originating from file %s line %d col %d\n",
            KnobMemAddrBeingOverwritten, appIP, fileName.c_str(), lineNum, column);
    printf (" writing %d bytes starting at %p\n", numBytesWritten, memWriteAddr);
}
Probe mode
Pin Probe-Mode

- Probe mode is a method of using Pin to instrument at the function level only. Wrap, Replace, call Analysis function before/after.

- Replacement or Wrapping function can call the replaced (original) function.

- The application and the replacement routine are run natively (not Jitted).
  - Faster than Jit-mode
  - Puts more responsibility on the tool writer.
  - Probes can only be placed on RTN boundaries
  - Must be inserted within the Image load callback.
  - Pin will automatically remove the probes when an image is unloaded.

- Many of the PIN APIs that are available in JIT mode are not available in Probe mode.
JIT Mode vs Probe Mode

• JIT Mode
  – Pin creates a modified copy of the application on-the-fly
  – Original code never executes

    ➢ More flexible, more common approach

• Probe Mode
  – Pin modifies the original application instructions
  – Inserts jumps to instrumentation code (trampolines)

    ➢ Lower overhead (less flexible) approach
A Sample Probe

- **A probe** is a jump instruction that overwrites original instruction(s) in the application
  - Instrumentation invoked with probes
  - Pin copies/translations original bytes so probed (replaced) functions can be called from the replacement function
A Sample Probe

0x400113d4: jmp 0x41481064
0x400113d5:
0x400113d7:
0x400113d8:
0x400113d9: push %ebx
...
...
0x41481064: ... // Tool code
...
...
0x414827fe: call 0x50000004 // Call orig func
...

0x50000004: push %ebp
0x50000005: mov %esp,%ebp
0x50000007: push %edi
0x50000008: push %esi
0x50000009: jmp 0x400113d9
PinProbes Instrumentation

• Advantages:
  – Low overhead – few percent
  – Less intrusive – execute original code
  – Leverages Pin:
    – API
    – Instrumentation engine

• Disadvantages:
  – More tool writer responsibility
  – Routine-level granularity (RTN)
Using Probes to Replace/Wrap a Function

- `RTN_ReplaceSignatureProbed()` redirects all calls to application routine `rtn` to the specified replacement function
  - Can add IARG_* types to be passed to the replacement routine, including pointer to original function and IARGCONTEXT.
  - Replacement function can call original function.

- To use:
  - Must use `PIN_StartProgramProbed()`
  - Application prototype is required
# Malloc Wrapping – Jit Mode

```c
#include "pin.H"

void * MallocWrapper( CONTEXT * ctxt, AFUNPTR pf_malloc, size_t size)
{
    // Simulate out-of-memory every so often
    void * res;
    if (TimeForOutOfMem())
        return (NULL);
    PIN_CallApplicationFunction(ctxt, PIN_ThreadId(),
                                CALLINGSTD_DEFAULT, pf_malloc,
                                PIN_PARG(void *), &res, PIN_PARG(size_t), size);

    return res;
}

VOID ImageLoad(IMG img, VOID *v) {
    // Pin callback. Registered by IMG_AddInstrumentFunction
    if (strstr(IMG_Name(img).c_str(), "libc.so") ||
        strstr(IMG_Name(img).c_str(), "MSVCR80") || strstr(IMG_Name(img).c_str(), "MSVCR90"))
    {
        RTN mallocRtn = RTN_FindByName(img, "malloc");

        PROTO protoMalloc = PROTO_Allocate( PIN_PARG(void *), CALLINGSTD_DEFAULT,
                                            "malloc", PIN_PARG(size_t), PIN_PARG_END() );

        RTN_ReplaceSignature(mallocRtn, AFUNPTR(MallocWrapper),
                             IARG_PROTOTYPE, protoMalloc,
                             IARG_CONST_CONTEXT,
                             IARG.ORIG_FUNCPTR,
                             IARG_FUNCARG_ENTRYPOINT_VALUE, 0,
                             IARG_END);
    }
}

int main(int argc, CHAR *argv[]) {
    PIN_InitSymbols();
    PIN_Init(argc,argv));
    IMG_AddInstrumentFunction(ImageLoad, 0);
    PIN_StartProgram();
}
```
#include "pin.H"

void * MallocWrapper(AFUNPTR pf_malloc, size_t size)
{
    // Simulate out-of-memory every so often
    void * res;
    if (TimeForOutOfMem())
        return (NULL);
    res = pf_malloc(size);
    return res;
}

VOID ImageLoad(IMG img, VOID *v) {
    if (strstr(IMG_Name(img).c_str(), "libc.so") ||
        strstr(IMG_Name(img).c_str(), "MSVCR80") || strstr(IMG_Name(img).c_str(), "MSVCR90"))
    {
        RTN mallocRtn = RTN_FindByName(img, "malloc");

        if ( RTN_Valid(mallocRtn) &&
            RTN_IsSafeForProbedReplacement(mallocRtn) )
        {
            PROTO proto_malloc = PROTO_Allocate(PIN_PARG(void *), CALLINGSTD_DEFAULT, "malloc",
                PIN_PARG(size_t), PIN_PARG_END() );

            RTN_ReplaceSignatureProbed(mallocRtn,
                AFUNPTR(MallocWrapper),
                IARG_PROTOTYPE, proto_malloc,
                IARG.ORIG_FUNCPTR,
                IARG_FUNCARG_ENTRYPOINT_VALUE, 0,
                IARG_END);
        }
    }
}

int main(int argc, CHAR *argv[])
{
    PIN_InitSymbols();    PIN_Init(argc, argv);
    IMG_AddInstrumentFunction(ImageLoad, 0);
    PIN_StartProgramProbed();
}
Using Probes to Call Analysis Functions

- *RTN_InsertCallProbed()* invokes the analysis routine before or after the specified `rtn`
  - Use `IPOINT_BEFORE` or `IPOINT_AFTER`
    - Pin may NOT be able to find all AFTER points on the function when it is running in Probe-Mode
    - PIN IARG_TYPEs are used for arguments

- To use:
  - Must use *PIN_StartProgramProbed()*
  - Application prototype is required for `IPOINT_AFTER`
Tool Writer Responsibilities

• No control flow into the instruction space where probe is placed
  – 5 bytes on IA-32, 7 bytes on Intel64
  – Branch into “replaced” instructions will fail
  – Probes at function entry point only

• Thread safety for insertion and deletion of probes
  – During image load callback is safe
  – Only loading thread has a handle to the image

• Replacement function has same behavior as original
The CONTEXT structure
CONTEXT*

- CONTEXT* is a **Handle** to the full register context of the application at a particular point in the execution
  - It can NOT be dereferenced.
  - It can only be passed to Pin API functions

- CONTEXT* is passed by default to a number of Pin Callback functions: **e.g.**
  - ThreadStart
    - Registered by `PIN_AddThreadStartFunction`
  - BufferFull
    - Registered by `PIN_DefineTraceBuffer`
  - OnContextChange
    - Registered by `PIN_AddContextChangeFunction`
CONTEXT*, IARG_CONST_CONTEXT, IARG_CONTEXT

- Pin provides API to Get and Set registers within the CONTEXT

- Can request CONTEXT* be passed to an analysis function by requesting IARG_(CONST)_CONTEXT

- Requesting IARG_CONTEXT
  - The analysis function will NOT be inlined
  - The passing of the CONTEXT* is time consuming

- Passing IARG_CONST_CONTEXT is ~4X faster than IARG_CONTEXT
  - Contents of CONTEXT* passed for IARG_CONST_CONTEXT can NOT be changed
• Changes made to the contents of a CONTEXT*
  – IARG_CONTEXT (Analysis routines)
    – Changes made will be visible in subsequent PIN API calls made from within the nesting of the analysis function
    – Changes made will NOT be visible in the application context after return from the analysis function
  – Passed to PIN Callbacks
    – Changes made will be visible also after callback returns
Function Replacement with register change

```c
#include "pin.H"

void *FunctionReplacer (  
    CONTEXT * ctx,  
    AFUNPTR pf_malloc, size_t size)
{
    void * res;  
    CONTEXT writableContext, * context = ctx;

    if (TimeForRegChange()) {
        PIN_SaveContext(ctx, &writableContext); // need to copy the ctx into a writable context  
        context = & writableContext;  
        PIN_SetContextReg(context , REG_GAX, 1);
    }
    PIN_CallApplicationFunction(context , PIN_ThreadId(), CALLINGSTD_DEFAULT, pf_malloc,  
        PIN_PARG(void *), &res, PIN_PARG(size_t), size);

    return res;
}

VOID ImageLoad(IMG img, VOID *v) {  // Pin callback. Registered by IMG_AddInstrumentFunction  
    RTN rtn = RTN_FindByName(img, "Function");
    PROTO proto = PROTO_Allocate( PIN_PARG(void *), CALLINGSTD_DEFAULT,  
        "proto", PIN_PARG(size_t), PIN_PARG_END() );

    RTN_ReplaceSignature (rtn, AFUNPTR(FunctionReplacer), IARG_PROTOTYPE, proto,  
        IARG_CONST_CONTEXT,  
        IARG_ORIG_FUNCPTR, IARG_FUNCARG_ENTRYPOINT_VALUE, 0, IARG_END);
}

int main(int argc, CHAR *argv[]) {  
    PIN_InitSymbols();  
    PIN_Init(argc,argv));  
    IMG_AddInstrumentFunction(ImageLoad, 0);  
    PIN_StartProgram();
}
Multi - Threading
Multi-Threading

• Pin fully supports multi-threading

  – Pin does not serialize application threads executing jitted code (including analysis code)
    – Pin provides synchronization constructs to introduce serialization if needed.
    – System calls require serialized entry to the VM before and after execution – BUT actual execution is NOT serialized

  – Pin does NOT create any threads of it’s own

  – Pin callbacks are serialized

  – *Jitting* is serialized
    – Only one application thread can be jitting code at any time
Multi-Threading services

• Pin Tools can:
  – Track Threads
    – ThreadStart, ThreadFini callbacks
    – IARG_THREAD_ID
  – Use Pin Virtual registers and TLS for thread-specific data
  – Use Pin Locks to synchronize threads
  – Create dedicated threads to do Pin Tool work
Using the TLS

- Pin tools can allocate TLS slots, by using the `PIN_CreateThreadDataKey()` function
  - Deallocate with `PIN_DeleteThreadDataKey()`

- Each thread can use `PIN_SetThreadData()` and `PIN_GetThreadData()` to access the TLS slots
  - Initial per-thread value is NULL

- Allocating a TLS slot receives an optional callback function
  - Callback will be invoked per thread upon thread exit, if the thread has a non-NULL value in the corresponding slot
Virtual registers

• Pin’s context structure includes several scratch general purpose registers
  – Do not map to actual architecture registers

• Can be accessed and modified same as physical registers

• Preferred to use the `PIN_ClaimToolRegister()` API
  – Claims a free scratch register to be used by the tool
  – Can help avoid contention when tool has several components which all require scratch registers
Pin Tool threads

• Pin tools may create their own threads
  – These threads will not be instrumented

• Use Pin API `PIN_SpawnInternalThread()`
  – System services, like `clone()` or `CreateThread()`, must not be used.

• Tool threads can only be created in the tool’s main(), or from within another tool thread
Locking Guidelines

• **Basic Rules**

  – Any locks acquired in a Pin callback, must be released before returning from that callback.

  – Any locks acquired in an analysis routine, must be released before returning from the analysis routine.

  – If the tool calls a Pin API from a callback, it should not hold any tool locks when calling the API.

  – If the tool calls a Pin API from an analysis routine, it should not hold any tool locks when calling the API
    – For some Pin API calls, the tool may need to acquire the Pin client lock first (see documentation of the API)
Locking Guidelines

• **Advanced Rules**
  - Some rules may be partially relaxed, in specific cases

  - If the tool acquires lock L in an analysis routine, it may continue holding L after the analysis routine completes:
    - Lock L must be released before leaving the trace that contains the analysis routine.
      - A trace may have multiple exit points
      - The tool must establish a callback which, in case of exception, releases the lock L. Tools can use `PIN_AddContextChangeFunction()` to establish this call-back.
    - Lock L may not be acquired by any Pin callback

  - The tool may hold a lock L while calling a Pin API, if that lock obeys the following sub-rule:
    - The tool does not acquire lock L from any call-back.
    - The Pin API invoked does not cause Application code to execute
Instrumenting a process tree
Instrumenting a Process Tree

• Process A creates Process B
  – Process B creates Process C and D
    – And so forth

• Pin can instrument all or part of the process tree
  – Use the \texttt{--follow\_exec} Pin invocation switch to turn this on
  – Can use different Pin modes (Jit or Probe) on the different processes in the process tree.
    – Can use different Pin Tools on the different processes of a process tree.

• Architecture of processes in the process tree may be intermixed
  – e.g. Process A is 32bit, Process B is 64 bit, Process C is 64 bit, Process D is 32 bit...
Instrumenting a Process Tree

// If this Pin Callback returns FALSE, then the child process will run Natively
BOOL FollowChild(CHILD_PROCESS childProcess, VOID * userData)
{
    BOOL res;
    INT appArgc;
    CHAR const * const * appArgv;

    OS_PROCESS_ID pid = CHILD_PROCESS_GetId(childProcess);

    // Get the command line that child process will be Pinned with, these are the Pin invocation switches
    // that were specified when this (parent) process was Pinned
    CHILD_PROCESS_GetCommandLine(childProcess, &appArgc, &appArgv);

    INT childArgc = 0;
    CHAR const * childArgv[20];
    [...] // :::: Create the Child’s Argc and Argv ::::

    CHILD_PROCESS_SetPinCommandLine(childProcess, childArgc, childArgv);

    return TRUE; /* Specify Child process is to be Pinned */
}

int main(INT32 argc, CHAR **argv)
{
    PIN_Init(argc, argv);
    cout << "Process is running on Pin in " << PIN_IsProbeMode() ? "Probe" : "Jit" " mode "

    // The FollowChild Callback will be called when the application is about to spawn a child process
    PIN_AddFollowChildProcessFunction (FollowChild, 0);

    if ( PIN_IsProbeMode() )
        PIN_StartProgramProbed(); // Never returns
    else
        PIN_StartProgram();
}
Part 3
optimizing Pin tools performance
Reducing Instrumentation Overhead

Total Overhead = Pin Overhead + Pintool Overhead

~5% for SPECfp and ~50% for SPECint
Pin team’s job is to minimize this

Usually much larger than pin overhead
Pintool writers can help minimize this!
Reducing the Pintool’s Overhead

Pintool’s Overhead

- Instrumentation Routines Overhead
- Analysis Routines Overhead

Frequency of calling an Analysis Routine × Work required in the Analysis Routine

Work required for transiting to Analysis Routine + Work done inside Analysis Routine
Tip #1
Reducing Work in Analysis Routines

• Key: Shift computation from analysis routines to instrumentation routines whenever possible

• This usually has the largest speedup
Counting control flow edges
void docount2(ADDRINT src, ADDRINT dst, INT32 taken) {
    COUNTER *pedg = Lookup(src, dst);
    pedg->count += taken;
}

void Instruction(INS ins, void *v) {
    if (INS_IsBranchOrCall(ins)) {
        INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)docount2,
            IARG_INST_PTR, IARG_BRANCH_TARGET_ADDR,
            IARG_BRANCH_TAKEN, IARG_END);
    }
}

...
void docount(COUNTER* pedge, INT32 taken) {
    pedge->count += taken;
}

void docount2(ADDRINT src, ADDRINT dst, INT32 taken) {
    COUNTER *pedg = Lookup(src, dst);
    pedg->count += taken;
}

void Instruction(INS ins, void *v) {
    if (INS_IsDirectBranchOrCall(ins)) {
        COUNTER *pedg = Lookup(INS_Address(ins),
                                INS_DirectBranchOrCallTargetAddress(ins));
        INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR) docount,
                       IARG_ADDRINT, pedg, IARG_BRANCH_TAKEN, IARG_END);
    }
    else if INS_IsIndirectBranchOrCall(ins) {
        INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR) docount2,
                       IARG_INST_PTR, IARG_BRANCH_TARGET_ADDR,
                       IARG_BRANCH_TAKEN, IARG_END);
    }
}
Tip #2
Reduce Analysis Calls Frequency

• Key: Instrument at the largest granularity whenever possible

• Instead of inserting one call per instruction, insert one call per basic block or trace
Slower Instruction Counting

counter++;
sub $0xff, %edx
counter++;
cmp %esi, %edx
counter++;
jle <L1>
counter++;
mov $0x1, %edi
counter++;
add $0x10, %eax
Faster Instruction Counting

Counting at BBL level

counter += 3
sub $0xff, %edx

cmp %esi, %edx
jle <L1>

counter += 2
mov $0x1, %edi
add $0x10, %eax

Counting at Trace level

sub $0xff, %edx

cmp %esi, %edx
jle <L1>

mov $0x1, %edi
add $0x10, %eax
counter += 5

counter += 3

L1
Tip #3
Reducing Work for Analysis Transitions

- Reduce number of arguments to analysis routines
- Inline analysis routines
- Use conditional instrumentation
- See how in next slides
Reduce Number of Arguments

• Eliminate arguments only used for debugging
• Instead of passing TRUE/FALSE, create 2 analysis functions
  - Instead of inserting a call to:
    Analysis(BOOL val)
  - Insert a call to one of these:
    AnalysisTrue()
    AnalysisFalse()

– IARG_CONTEXT is very expensive (> 10 arguments)
  – Use the cheaper IARG_CONST_CONTEXT
### Inlining

#### Inlinable

```c
int docount0(int i) {
    x[i]++
    return x[i];
}
```

#### Not-inlinable

```c
int docount1(int i) {
    if (i == 1000)
        x[i]++;
    return x[i];
}
```

```c
int docount2(int i) {
    x[i]++;
    printf("%d", i);
    return x[i];
}
```

```c
void docount3() {
    for(i=0;i<100;i++)
        x[i]++;
}
```

#### Not-inlinable
Inlining

• Use the `–log_inline` invocation switch to record inlining decisions in `pin.log`

```bash
pin –log_inline –t mytool – app
```

• Look in `pin.log`

  Analysis function (0x2a9651854c) from mytool.cpp:53  INLINED

  Analysis function (0x2a9651858a) from mytool.cpp:178 NOT INLINED
  
  The last instruction of the first BBL fetched is not a ret instruction

• Look at source or disassembly of the function in `mytool.cpp` at line 178

```assembly
0x00000002a9651858a push rbp
0x00000002a9651858b mov rbp, rsp
0x00000002a9651858e mov rax, qword ptr [rip+0x3ce2b3]
0x00000002a96518595 inc dword ptr [rax]
0x00000002a96518597 mov rax, qword ptr [rip+0x3ce2aa]
0x00000002a9651859e cmp dword ptr [rax], 0xf4240
0x00000002a965185a4 jnz 0x11

  The function could not be inlined because it contains a control-flow changing instruction (other than ret)
```
Conditional instrumentation

- A common scenario where the analysis routine has a single “if-then”
  - The “If” part is always executed
  - The “then” part is rarely executed
  - Useful cases:
    1. “If” can be inlined, “Then” is not
    2. “If” has small number of arguments, “then” has many arguments (or IARG_CONTEXT)

- Pintool writer breaks analysis routine into two:
  - `INS_InsertIfCall (ins, ..., (AFUNPTR)doif, ...)`
  - `INS_InsertThenCall (ins, ..., (AFUNPTR)dothen, ...)`
VOID Instruction(INS ins, VOID *v) {
    INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)IpSample,
                   IARG_INST_PTR, IARG_END);
}

VOID IpSample(VOID* ip) {
    --icount;
    if (icount == 0) {
        fprintf(trace, "%p\n", ip);
        icount = N + rand()%M; //icount is between <N, N+M>
    }
}

const INT32 N = 10000; const INT32 M = 5000;
INT32 icount = N;
INT32 CountDown() {
    --icount;
    return (icount==0);
}

VOID PrintIp(VOID *ip) {
    fprintf(trace, "%p\n", ip);
    icount = N + rand()%M; //icount is between <N, N+M>
}

VOID Instruction(INS ins, VOID *v) {
    // CountDown() is always called before an inst is executed
    INS_InsertIfCall(ins, IPOINT_BEFORE, (AFUNPTR)CountDown,
                     IARG_END);

    // PrintIp() is called only if the last call to CountDown() returns a non-zero value
    INS_InsertThenCall(ins, IPOINT_BEFORE, (AFUNPTR)PrintIp,
                        IARG_INST_PTR, IARG_END);
}
Jitting time

- Jitting is expensive
  - Takes far more time to jit an instruction than to execute a jitted instruction

- Portions of a workload where very many IPs are being jitted, and executed a small number of times
  - Jitting time dominates execution time
  - E.g.
    - startup of a GUI app
    - Compiler compiling a non-large file
  - Vs Loop executing a large number of times
    - Jitting time is amortized over execution time
Optimizing Your Pintools - Summary

• Baseline Pin has fairly low overhead for non-jitting portions of workloads (~5-20%)

• Adding instrumentation can increase overhead significantly, but you can help!

1. Move work from analysis to instrumentation routines
2. Explore larger granularity instrumentation
3. Explore conditional instrumentation
4. Understand when Pin can inline instrumentation
Part 4
Advanced Pin

OS Specifics: Windows
OS Specifics: Linux
Managing exceptions
Managing signals
Code Cache API
Debugging & Pin
OS Specifics - Windows
Windows Challenges (1/2)

• **Handling system calls**
  – Pin must intercept system calls to regain control of the application on return from the system
  – Pin must monitor system calls to notify instrumentation when DLLs are loaded/unloaded, threads are created/terminated, etc.
    – **System call interface is undocumented**
    – **System call numbers potentially change with each system build**

• **Handling exceptions and asynchronous interruptions**
  – To maintain control and notify instrumentation about control flow changes Pin must intercept all transitions from kernel to user mode
    – **Windows is not designed to have an independent agent interposed between the kernel and application**
      – The kernel dispatches interruptions via (undocumented) entry points in ntdll.dll
Windows Challenges (2/2)

**Injection**
- PIN VMM is a DLL that must be loaded into the address space of the application to get initial control of the process
- **Windows is not designed for proprietary loader**
- Common practice: intercept control at the entry point of the application
  - Instrumentation can not observe initialization procedures in statically linked application DLLs
  - Injection presented in the introduction is referred to as Late Injection
    - It misses the initialization procedures in statically linked application DLLs
- **Early injection is not trivial**

**Isolation of instrumentation from the application**
- Instrumentation runs in the same process as the application it is observing
- Enabling C run-time in the instrumentation causes sharing of system libraries (e.g. kernel32.dll) and their state with the application
- To be transparent, Pin must
  - *Preserve original state of system resources*
  - *Avoid reentrant use of shared libraries*
**Injection**

- Injection is the procedure for loading the PINVM.DLL into the address space of an application and gaining control of execution

- Other systems hook the entry point of the application
  - Too late: initialization procedures in application DLLs can not be instrumented

- For maximum observability, Pin should inject itself into a new process as early as possible, however...

- Pin depends on some basic system services so it is not possible to load PINVM.DLL until the loader and kernel32.dll have initialized

- **The optimal injection point: just after initialization of kernel32.dll**
  - Injection presented in the introduction is referred to as Late Injection
    - It misses the initialization procedures in statically linked application DLLs
Early Injection step by step

```
pin -t pintool.dll -- application.exe
```

### Pin Boot Routine

- Create (suspended) application process
- Attach to the application as a debugger
- Run the application process until kernel32.dll is loaded and initialized
- Detach from the application process
- Copy Boot Routine into the application process and set PC to start of the routine
- Resume application process

### PIN.EXE

- Load and start Pin VMM

### PINVM.DLL

- Load the instrumentation tool
- Instrument and execute the application

All application instructions are executed under Pin control.
Handling System Calls

- **Pin must manage the execution of system calls**
  - To regain control when the system returns to user mode with a modified thread context
  - To monitor and handle some important system events
    - Loading DLLs, creation and termination of threads and processes, etc.

- **Pin intercepts system call instructions, not Win32 APIs**
  - Pin instruments all modules in the user space, including system libraries
  - Some applications use native API (NTDLL interface) directly, bypassing Win32 API
  - Win32 API layer is very wide, while system call instructions are easy to discover

- **Three steps in managing system calls:**
  - Detect a system call and redirect control to VMM
  - Execute the system call on behalf of the application
  - Regain control when the kernel returns to user with a new context
    - The system may interrupt system call execution by asynchronous calls to application procedures
System Call Interception

- Pin detects system call instructions when it generates traces in the code cache
  - IA-32: *sysenter* and *int 2E*; Intel64: *syscall*; etc.
  - This is a static analysis, so the overhead is low

- Pin executes system calls in VMM, not in the code cache - emits jump to VMM instead of the system call instruction
  - Enables flushing the code cache while a system call blocks in the kernel
  - VM lock is NOT held during the actual syscall

- Some system calls may affect Pin’s internal state. To handle them properly, Pin must know the corresponding system call numbers
  - Windows system call numbers are unpublished and potentially change with each system build
  - Pin discovers system call numbers dynamically, on the early stage of the injection process
    - *We trace the corresponding NTDLL functions until a system call instruction is reached and then read the system call number from the EAX/RAX register*
System Call Execution

• The **System Call Emulator** executes all “known” system calls that may affect the VMM state, e.g. memory mappings, creation and termination of threads and processes, etc.

• The remaining, unknown system calls are forwarded to the **System Gate**
  – Per-thread procedure that transparently executes system calls and regains control upon return or interruption
  – Fills/spills original context before/after system calls
  – Recovers original context (PC) when a system call is interrupted

System Gate executes system calls “blindly”, assuming that each of them can arbitrarily modify context and control flow (if interrupted)
User Procedure Calls (UPC)

- UPC is a control transfer from the kernel to a user-level procedure

**Asynchronous procedure call (APC)**
- Asynchronous events: file I/O completion, timer expiration
- Thread initialization APC signals start of a new thread

**Callback**
- Asynchronous Windows GUI message

**Exception**
- Access violation, illegal instruction, divide by zero, etc.

Asynchronous events are not delivered immediately, but wait in queue until the application invokes an *interruptible (alertable)* system call

**Pin must intercept UPCs to maintain control of the application and recover the original interruption context (visible to the application)**
UPC Interception

- The kernel dispatches UPCs through entry points in NTDLL.DLL
- To intercept UPCs, Pin overwrites the NTDLL entry points with trampolines that jump to the *Event Dispatcher* in Pin
- When a UPC is intercepted, Pin recovers original interruption context in the UPC frame prepared by the kernel
  - *JIT Compiler* recovers context of exceptions that occurred in the code cache
  - *System Call Emulator* recovers context of interrupted system calls

![Diagram showing UPC interception process]

Pin intercepts all control transfers from the kernel to the user mode
Exceptions (1/2)

• Unlike APCs and callbacks that are queued and delivered at the next alertable system call, exceptions are synchronous events.

• Exceptions do not necessarily cause abnormal termination of the process – the application may expect and handle exceptions.

• Pin must provide exception handlers with the same exception information that accompanies exceptions in the native application:
  – Exception context, code and exception-specific parameters.

• From Pin’s perspective, there are three kinds (sources) of exceptions in Windows applications:
  – An attempt to fetch an invalid or inaccessible instruction.
  – An attempt to execute a faulting instruction.
  – Software exceptions generated by the application.
Exceptions (2/2)

• Decoder (fetcher) of instructions raises an exception if it encounters an invalid or inaccessible instruction
  - When the kernel delivers this exception back to the user mode, Pin skips the context translation because it sees original PC in the exception context

• Other **hardware exceptions** that occur in the code cache
  - Recovery of the original exception context is nontrivial due to register allocation
  - Pin retranslates the interrupted trace to get the virtual-physical register bindings at the faulting point
  - Optimization: small cache of register bindings for frequent exceptions

• Other **hardware** exceptions that occur in the tool code
  - Pin APIs for tool to manage it’s exceptions

• Application can generate **software exceptions** using Win32 API
  - The exception context represents an original application state
  - Context translation is not needed

Pin delivers precise exceptions to applications handlers
Multithreading Support

• Pin instruments and runs all threads of the application from the first to the last user-mode instruction
  – Attaches to a new thread when the system delivers the thread initialization APC
  – Maintains control until the thread exits
  – Intercepts threads created by remote processes

• Pin’s threading activities are transparent to the application
  – Pin VMM serializes some of its operations (e.g. JIT compilation), but never executes code of the application under Pin locks
  – Except for initialization phase, Pin never acquires windows locks in system libraries, e.g. loader lock or process heap lock
  – Each thread has a shadow stack that is used by Pin VMM and Pintool
Thread-Local State

• Key elements of Pin’s thread-local state:
  – **Spill area** keeps values of spilled virtual registers
    – JIT-compiled traces need fast access to spilled register values
    – Pin “steals” one physical register to point to the spilling area
  – **TEB (Thread Environment Block) state**
    – Keeps original thread-local state of system libraries, e.g. last Win32 error value, stack limit
    – C run-time routines may access/modify these values
    – Need to preserve the original state while running in Pin VMM or PinTool
  – **System call state**
    – Contains information about active and interrupted system calls in the thread
    – The information is used to restore the original context on return from the system

• Pin steals one TLS slot from the application to enable fast access to the thread-local data in Pin VMM
Thread Suspension and Context Manipulation

- A thread can suspend another thread and read/modify its context
  - `SuspendThread()`, `GetThreadContext()`, `SetThreadContext()`

- Pin must emulate the corresponding system calls to avoid deadlocks and transparency issues
  - Target thread may hold a Pin lock
  - The thread context is not original
  - Suspended traces disable flushing the code cache

**Solution:** Force a thread to leave the code cache and wait until the thread reaches a *safe point*

*Safe point = no locks, not in the code cache, accessible original context*

- Unlink the suspended trace from successors and let it enter VMM
- Block the thread in the *safe VMM point* or in the *System Gate*
- Use thread-local data to store and access the original context associated with the safe point
Isolation (1/2)

- Pin Tools are compiled to use the static CRT

- Pin on Windows does not separate DLLs loaded by the tool from the application DLLs - it uses the same system loader.

  - The tool should not load any DLL that can be shared with the application.

  - The tool should avoid static links to any common DLL, except for those listed in PIN_COMMON_LIBS (see source\tools\ms.flags file).
Isolation (2/2)

• Pin on Windows guarantees safe usage of C/C++ run-time services in Pin tools, including indirect calls to Windows API through C run-time library.
  
  – Any other use of Windows API in Pin tool is not guaranteed to be safe

• Pin uses some base types that conflict with Windows types. If you use "windows.h", you may see compilation errors. So do:

  namespace WINDOWS { #include <windows.h> }
OS Specifics - Linux
Linux Challenges (1/2)

• Handling system calls
  – Pin must intercept system calls to regain control of the application on return from the system
  – Pin must monitor system calls to notify instrumentation when DLLs are loaded/unloaded, threads are created/terminated, etc.
  – **Some system calls may behave differently on different Linux distributions.**

• Signal handling
  – Pin must identify whether the signal originated from the application, the tool or Pin itself.
  – **Pin cannot seem to interfere with the applications signal mask.**
Linux Challenges (2/2)

• **Injection**
  - Pin relies on the ptrace system call for injection.
  - Some platforms do not allow tracing a parent application by a child. In such cases the application is run on the child.

• **Isolation of instrumentation from the application**
  - Instrumentation runs in the same process as the application it is observing.
  - Pin must emulate several libc services.
fork

Child (Injector):

exitLoop = FALSE;
Ptrace TraceMe
while(!exitLoop){}

Ptrace Injectee – Injectee Freezes

Injectee.exitLoop = TRUE;

Ptrace continue (unFreezes Injectee)

execv(gzip);
// Injectee Freezes
Execution of Injector resumes after execv(gzip) in Injectee completes

Ptrace Copy (save, gzip.CodeSegment, sizeof(MiniLoader))
PtraceGetContext (gzip.OrigContext)
PtraceCopy (gzip.CodeSegment, MiniLoader, sizeof(MiniLoader))

Ptrace continue@MiniLoader (unFreezes Injectee)

MiniLoader loads Pin+Tool, allocates Pin stack
Kill(SigTrace, Injector): Freezes until Ptrace Cont

Wait for MiniLoader complete (SigTrace from Injectee)

Ptrace Copy (gzip.CodeSegment, save, sizeof(MiniLoader))
Ptrace Copy (gzip.pin.stack, gzip.OrigCtxt, sizeof (ctxt))
Ptrace SetContext (gzip.IP=pin, gzip.SP=pin.Stack)

Ptrace Detach

MiniLoader

Pin Code and Data

gzip (Injectee)

Pin stack

Code to Save

gzip Code and Data

MiniLoader

Pin Code and Data

Inscount2.so

IP

gzip OrigCtxt

Pin Code and Data
Handling System Calls

- **Pin must manage the execution of system calls**
  - Pin must maintain control all the time
  - System calls are executed inside pin and return to the application
  - In most cases the system call is executed without the pin VM lock
  - Certain system calls are emulated by pin (see below)

- **System call emulation**
  - Pin detects if a system call needs emulation.
  - Pin needs to know the attributes of each memory page for SMC support
    - Therefore all system calls related to memory are emulated by pin
  - Signal related system calls are emulated
  - Creating of new threads and new child processes
  - Setting/getting of the TLS segment registers
  - Thread and process termination
**Signal Handling**

- Pin registers its own signal handlers for all signals, and saves the application’s handlers.
- Pin must handle both synchronous and asynchronous signals.

**Asynchronous signals:**
- These signals may be delivered “at will” so Pin waits for safe point to deliver them.
- When such a signal arrives, Pin’s internal handler registers this signal, unlinks the current trace and resumes execution from the code cache.
- At the trace’s exit point, the executing thread jumps to the VM, thus transferring control over to Pin. The VM checks if there are pending signals and calls the application’s original signal handlers for these signals (jitting them).

**Synchronous signals:**
- These signals must be delivered immediately.
- They may originate from the application, the tool or Pin itself.
- Pin’s internal handler is called, it determines the origin of the signal and propagates the signal delivery to the tool and application if necessary.
- If signal is delivered to the application, the application’s signal handler is jitted.
Multithreading Support

- Pin instruments and runs all threads of the application from the first to the last user-mode instruction
  - Attaches to the thread upon the first user-space instruction
  - Maintains control until the thread exits

- Pin’s threading activities are transparent to the application
  - The Pin VM serializes some of its operations (e.g. JIT compilation), but never executes code of the application under Pin locks
  - Each thread has a shadow stack that is used by the Pin VM and the Pintool
  - Pin and pintools are prohibited from using the pthread library due to conflicts with some internal structures. Therefore Pin provides its own APIs for thread creation and control.
Thread-Local Storage

• Segment virtualization
  – TLS is accessed via the fs (64 bit) or gs (32 bit) segment register.
  – Both the application and Pin share this register, but expect different values.
  – Pin emulates the application’s usage of the fs/gs register thus isolating the application’s TLS for Pin’s.
Isolation (1/2)

- Pin is injected into address space and has its own copy of the dynamic loader and runtime libraries (GLIBC, etc).

- Pin uses a small library of CRT for direct calls to system calls.

- The process has a single signals table (shared among all threads), pin manages an internal signal table and emulate all the system calls related to signals.
Isolation (2/2)

- pthread functions cannot be called from an analysis or replacement routine

- Pintools on Linux need to take care when calling standard C or C++ library routines from analysis or replacement functions
  - because the C and C++ libraries linked into Pintools are **not** thread-safe
Managing Exceptions
Exceptions

• Catch Exceptions that occur in Pin Tool code

  – Global exception handler
    – *PIN_AddInternalExceptionHandler()*

  – Guard code section with exception handler
    – *PIN_TryStart()*
    – *PIN_TryEnd()*
VOID InstrumentDivide(INS ins, VOID* v)
{
    if ((INS_Mnemonic(ins) == "DIV") &&
        (INS_OperandIsReg(ins, 0)))
    {
        // Will Emulate div instruction with register operand
        INS_InsertCall(ins,
                       IPOINT_BEFORE,
                       AFUNPTR(EmulateIntDivide),
                       IARG_REG_REFERENCE, REG_GDX,
                       IARG_REG_REFERENCE, REG_GAX,
                       IARG_REG_VALUE, REG(INS_OperandReg(ins, 0)),
                       IARG_CONST_CONTEXT,
                       IARG_THREAD_ID,
                       IARG_END);
        INS_Delete(ins); // Delete the div instruction
    }
}

int main(int argc, char * argv[])
{
    PIN_Init(argc, argv);
    INS_AddInstrumentFunction (InstrumentDivide, 0);
    PIN_AddInternalExceptionHandler (GlobalHandler, NULL); // Registers a Global Exception Handler
    PIN_StartProgram(); // Never returns
    return 0;
}
VOID EmulateIntDivide(ADDRINT * pGdx, ADDRINT * pGax, ADDRINT divisor, CONTEXT * ctxt, THREADID tid)
{
    PIN_TryStart(tid, DivideHandler, ctxt); // Register a Guard Code Section Exception Handler
    UINT64 dividend = *pGdx;
dividend <<= 32;
dividend += *pGax;
*pgax = dividend / divisor;
*pGdx = dividend % divisor;
PIN_TryEnd(tid); /* Guarded Code Section ends */
}
Exceptions example (3/3)

EXCEPT_HANDLING_RESULT
DivideHandler (THREADID tid, EXCEPTION_INFO * pExceptInfo,
        PHYSICAL_CONTEXT * pPhysCtxt, // The context when the exception occurred
        VOID *appContextArg ) // The application context when the exception occurred
{
    if(PIN_GetExceptionCode(pExceptInfo) == EXCEPTCODE_INT_DIVIDE_BY_ZERO)
    {
        // Divide by zero occurred in the code emulating the divide, use PIN_RaiseException to raise this
        // exception at the appIP – for handling by the application
        cout << " DivideHandler : Caught divide by zero. " << PIN_ExceptionToString(pExceptInfo) << " endl; 

        // Get the application IP where the exception occurred from the application context
        CONTEXT * appCtxt = (CONTEXT *)appContextArg;
        ADDRINT faultIp = PIN_GetContextReg (appCtxt, REG_INST_PTR);

        // raise the exception at the application IP, so the application can handle it as it wants to
        PIN_SetExceptionAddress (pExceptInfo, faultIp);
        PIN_RaiseException (appCtxt, tid, pExceptInfo); // never returns
    }
    return EHR_CONTINUE_SEARCH;
}

EXCEPT_HANDLING_RESULT
GlobalHandler(THREADID threadIndex, EXCEPTION_INFO * pExceptInfo,
        PHYSICAL_CONTEXT * pPhysCtxt, VOID *v)
{
    cout << "GlobalHandler: Caught unexpected exception. " << PIN_ExceptionToString(pExceptInfo) << " endl; 
return EHR_UNHANDLED;
}
Monitoring Application Exceptions

- `PIN_AddContextChangeFunction()`
  - Can monitor and change that application state at application exceptions

```c
int main(int argc, char **argv)
{
    PIN_Init(argc, argv);

    PIN_AddContextChangeFunction(OnContextChange, 0);

    PIN_StartProgram();
}
```
Monitoring Application Exceptions

static void OnContextChange (THREADID tid,
    CONTEXT_CHANGE_REASON reason,
    const CONTEXT *ctxtFrom  // Application's register state at exception point
    CONTEXT *ctxtTo,         // Application's register state delivered to handler
    INT32 info,
    VOID *v)
{
    if (CONTEXT_CHANGE_REASON_SIGRETURN == reason
        || CONTEXT_CHANGE_REASON_APC == reason
        || CONTEXT_CHANGE_REASON_CALLBACK == reason
        || CONTEXT_CHANGE_REASON_FATALSIGNAL == reason
        || ctxtTo == NULL) // don't want to handle these
        return;

    // CONTEXT_CHANGE_REASON_EXCEPTION
    // change some register values in the context that the application will see at the handler
    FPSTATE fpContextFromPin;
    // change the bottom 4 bytes of xmm0
    PIN_GetContextFPState (ctxtFrom, &fpContextFromPin);
    fpContextFromPin.fsave_legacy._xmm[3] = 'de';
    fpContextFromPin.fsave_legacy._xmm[2] = 'ad';
    fpContextFromPin.fsave_legacy._xmm[1] = 'be';
    fpContextFromPin.fsave_legacy._xmm[0] = 'ef';
    PIN_SetContextFPState (ctxtTo, &fpContextFromPin);

    // change eax
    PIN_SetContextReg(ctxtTo, REG_RAX, 0xbaadf00d);
}
Managing Signals
Signals

• Tools can establish an interceptor function for signals delivered to the application
  – Tools should never call sigaction() directly to handle signals.
  – Interceptor function is called whenever the application receives the requested signal, regardless of whether the application has a handler for that signal.
  – Interceptor function can then decide whether the signal should be forwarded to the application
Signals

• A tool can take over ownership of a signal in order to:
  – use the signal as an asynchronous communication mechanism to the outside world.
    – For example, if a tool intercepts SIGUSR1, a user of the tool could send this signal and tell the tool to do something. In this usage model, the tool may call PIN_UnblockSignal() so that it will receive the signal even if the application attempts to block it.
  – "squash" certain signals that the application generates.
    – a tool that forces speculative execution in the application may want to intercept and squash exceptions generated in the speculative code.

• A tool can set only one "intercept" handler for a particular signal, so a new handler overwrites any previous handler for the same signal. To disable a handler, pass a NULL function pointer.
Signals

BOOL EnableInstrumentation = FALSE;

BOOL SignalHandler(THREADID, INT32, CONTEXT *, BOOL, const EXCEPTION_INFO *, void *)
{
    // When tool receives the signal, enable instrumentation. Tool calls
    // PIN_RemoveInstrumentation() to remove any existing instrumentation from Pin's code cache.
    EnableInstrumentation = TRUE;
    PIN_RemoveInstrumentation();

    return FALSE; /* Tell Pin NOT to pass the signal to the application. */
}

VOID Trace(TRACE trace, VOID *)
{
    if (!EnableInstrumentation)
        return;

    for (BBL bbl = TRACE_BblHead(trace); BBL_Valid(bbl); bbl = BBL_Next(bbl))
        BBL_InsertCall(bbl, IPOINT_BEFORE, AFUNPTR(AnalysisFunc), IARG_INST_PTR, IARG_END);
}

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

    PIN_InterruptSignal(SIGUSR1, SignalHandler, 0); // Tool should really determine which signal is NOT in
    // use by application

    PIN_UnblockSignal(SIGUSR1, TRUE);
    TRACE_AddInstrumentFunction(Trace, 0);

    PIN_StartProgram();
}
Code-Cache API
The Code-Cache API allows a Pin Tool to:
- Inspect Pin's code cache and/or alter the code cache replacement policy
- Assume full control of the code cache
- Remove all or selected traces from the code cache
- Monitor code cache activity, including start/end of execution of code in the code cache
VOID DoSmcCheck(VOID * traceAddr, VOID * traceCopyAddr, USIZE traceSize, CONTEXT * ctxP) {
    if (memcmp(traceAddr, traceCopyAddr, traceSize) != 0) /* application code changed */
    {
        // the jitted trace is no longer valid
        free(traceCopyAddr);
        CODECACHE_InvalidateTraceAtProgramAddress((ADDRINT)traceAddr);
        PIN_ExecuteAt(ctxP); /* Continue jited execution at this application trace */
    }
}

VOID InstrumentTrace(TRACE trace, VOID *v) {
    VOID * traceAddr; VOID * traceCopyAddr; USIZE traceSize;

    traceAddr = (VOID *)TRACE_Address(trace); // The appIP of the start of the trace
    traceSize = TRACE_Size(trace); // The size of the original application trace in bytes
    traceCopyAddr = malloc(traceSize);

    if (traceCopyAddr != 0)
    {
        memcpy(traceCopyAddr, traceAddr, traceSize); // Copy of original application code in trace
        // Insert a call to DoSmcCheck before every trace
        TRACE_InsertCall(trace, IPOINT_BEFORE, (AFUNPTR)DoSmcCheck,
                         IARG_PTR, traceAddr,
                         IARG_PTR, traceCopyAddr,
                         IARG_UINT32 , traceSize,
                         IARG_CONTEXT,
                         IARG_END);
    }
}

int main(int argc, char * argv[]) {
    PIN_Init(argc, argv);
    TRACE_AddInstrumentFunction(InstrumentTrace, 0);
    PIN_StartProgram();
}
Debugging & Pin
Transparent debugging, and extending the debugger

• Transpareently debug the application while it is running on Pin + Pin Tool
  – **PinADX**: Customizable Debugging with Dynamic Instrumentation (Presented at CGO 2012)

• Use Pin Tool to enhance/extend the debugger capabilities
  – Watchpoint: Is order of magnitude faster when implemented using Pin Tool
  – Which branch is branching to address 0
    – Easy to write a Pin Tool that implements this
Debug Application while Running Pin

• Useful for Pin-based emulators
  – User can debug application while emulating

• Provide advanced debugging features with Pin:
  – Stack monitoring features
  – Buffer overrun detection
  – Reverse debugging
  – Write your own debugger extension via Pin
Naïve Solution Won’t Work

- Why can’t we just debug normally?
  - Debugger sees Pin state, not application state
  - Pin recompiles application code
  - Instructions wrong, registers wrong, PC wrong, ...

![Diagram showing GDB, Pin, Application, and Tool with arrows and question mark]
Pin Debugger Interface

- GDB debugs application (not Pin itself)
- Leverage GDB remote protocol ABI
1. Run Pin with -appdebug

```
$ pin -appdebug -t tool.so -- ./application
Application stopped until continued from debugger.
Start GDB, then issue this command at the (gdb) prompt:
target remote :1234
```

2. Start GDB, enter “target remote …”

```
$ gdb ./application
(gdb) target remote :1234
```

3. Set breakpoints, etc. Continue with “cont”

```
(gdb) break main
(gdb) cont
```
Extending the Debugger

• Normal debugging with Pin useful but limited

• Extending the debugger:
  – Add GDB commands via a Pin tool
  – Stop at “semantic breakpoint” via instrumentation

• Use the “monitor” keyword for implementing custom commands
Stack Debugger Pintool

$ pin -appdebug -t stack-debugger.so -- ./app

$ gdb ./app
(gdb) target remote :1234
(gdb) monitor stackbreak 4000
Break when thread uses 4000 stack bytes.
(gdb) cont
Thread uses 4004 stack bytes.
[...] 
(gdb) monitor stats
Maximum stack usage: 8560 bytes.

Commands implemented in Pintool
Stack-Debugger Instrumentation

• Thread Start:
  \( \text{StackBase} = \%\text{esp}; \)
  \( \text{MaxStack} = 0; \)

• [...] sub $0x60, \%\text{esp}
  size = StackBase - \%\text{esp};
  if (size > MaxStack) MaxStack = size;
  if (size > StackLimit) TriggerBreakpoint();

• cmp \%esi, \%edx
  jle <L1>

After each stack-changing instruction
VOID Instruction(INS ins, VOID *)  
   {
      if (INS_RegWContain(ins, REG_STACK_PTR))
      {
         IPOINT where = (INS_HasFallThrough(ins)) ?
            IPOINT_AFTER : IPOINT_TAKEN_BRANCH;
         INS_InsertCall(ins, where, (AFUNPTR)OnStackChange,
            IARG_REG_VALUE, REG_STACK_PTR,
            IARG_THREAD_ID, IARG_CONST_CONTEXT, IARG_END);
      }
   }

VOID OnStackChange(ADDRINT sp, THREAD ID tid, CONTEXT *ctxt)
   {
      size_t size = StackBase - sp;
      if (size > StackMax) StackMax = size;
      if (size > StackLimit) {
         ostringstream os;
         os << "Thread uses " << size << " stack bytes.";
         PIN_ApplicationBreakpoint(ctxt, tid, FALSE, os.str());
      }
   }
int main() {
    [...]  
    PIN_AddDebugInterpreter(HandleDebugCommand, 0);
}

BOOL HandleDebugCommand(const string &cmd, string *result) {
    if (cmd == "stats") {
        ostringstream os;
        os << "Maximum stack usage: " << StackMax << " bytes.\n";
        *result = os.str();
        return TRUE;
    }
    else if (cmd.find("stackbreak ") == 0) {
        StackLimit = /* parse limit */;
        ostringstream os;
        os << "Break when thread uses " << StackLimit << " stack bytes.";
        *result = os.str();
        return TRUE;
    }
    return FALSE;  // Unknown command
}
Other Debugger Tools

- Breakpoint on buffer overrun
- Debug from a recorded log file
- Reverse debugging from a recording
- Design your own custom debugger tool
Summary

• Pin is Intel’s dynamic binary instrumentation engine

• Pin can be used to instrument all user level code
  – Windows, Linux, OSX, Android
  – IA-32, Intel64
  – Product level robustness
  – Jit-Mode for full instrumentation: Thread, Function, Trace, BBL, Instruction
  – Probe-Mode for Function Replacement/Wrapping/Instrumentation only.
  – Pin supports multi-threading, no serialization of jitted application nor of instrumentation code

• Pin API makes Pin tools easy to write
  – Presented many tools, many fit on 1 ppt slide

• Pin performance is good
  – Pin APIs provide for writing efficient Pin tools

• Popular and well supported
  – 30,000+ downloads, 700+ citations

• Free Download
  – www.pintool.org
  – Includes: Detailed user manual, source code for 100s of Pin tools, tutorials

• Pin User Group
  – http://tech.groups.yahoo.com/group/pinheads/
  – Pin users and Pin developers answer questions
Final note

- Use the Pin manual!
  - www.pintool.org -> User’s manual

- A lot more information about using Pin

- Many more topics – beyond this tutorial
  - How to debug your Pin tool
  - Trace buffers
  - System calls instrumentation
  - Instruction decoding APIs (XED)
  - ... And many others
Now go and write your Pin tools!