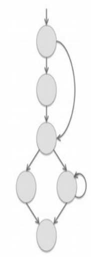
Dynamic Analysis: Introduction to dynamic binary instrumentation and Intel Pin

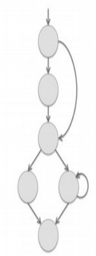
Sanjay Rawat



Call and control flow graphs

- First, disassemble
- Valid `call` targets become function entry points
- For each function, perform a breadth-first control flow graph traversal
- Once function ranges/basic blocks are identified, we get a call graph

- This procedure might leave us with gaps in the text space that were not analyzed!
  - Functions that are never statically referenced in previously visited code
  - Fragments of functions never statically referenced in previously visited code
  - The presence of data
  - Alignment padding



Call and control flow graphs

- First, disassemble
- Valid `call` targets become function entry points
- For each function, perform a breadth-first control flow graph traversal
- Once function ranges/basic blocks are identified, we get a call graph

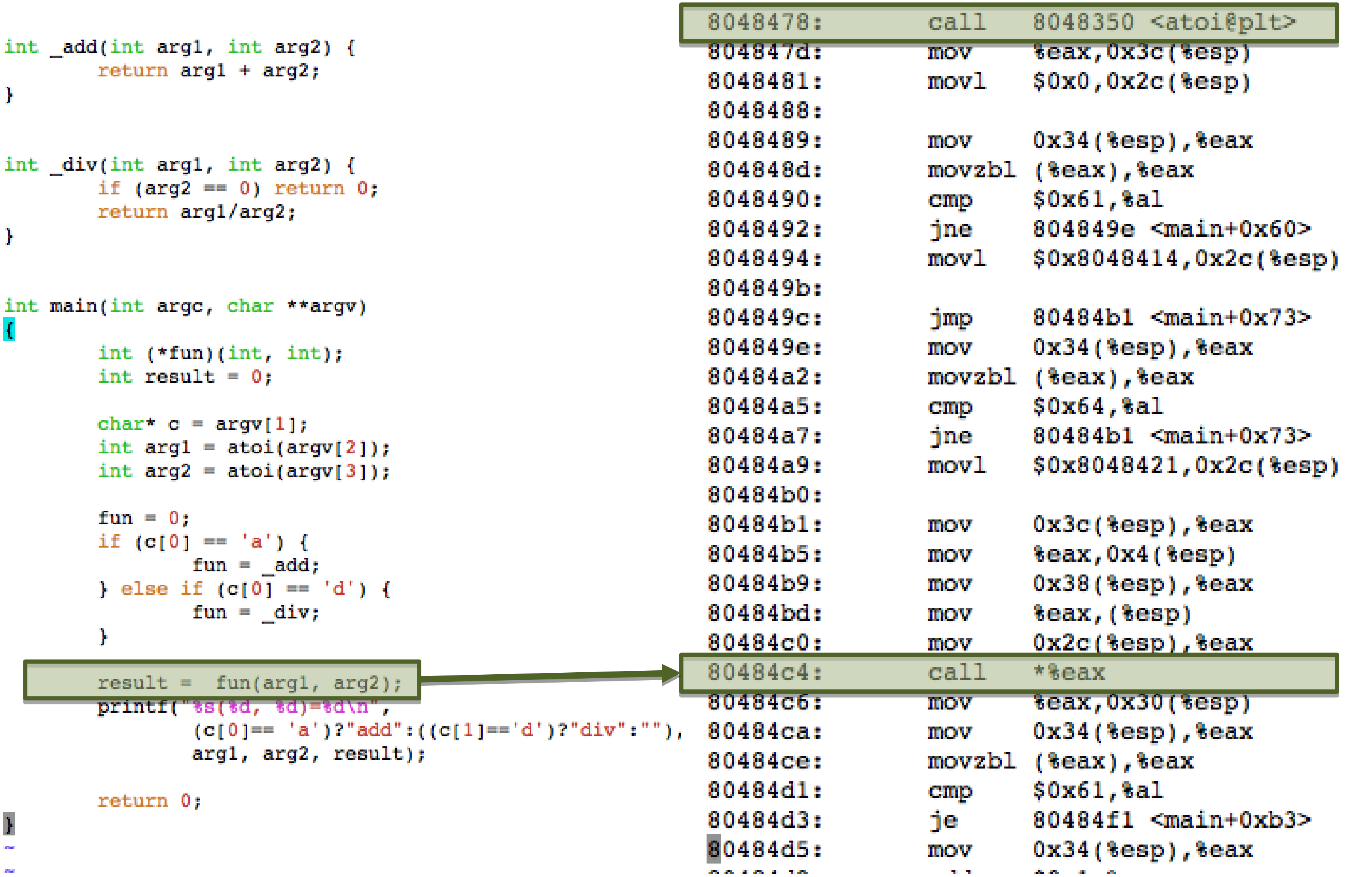
- This procedure at were not analyzed!
  - Functions that a ode
  - Fragments of fu ted code
  - The presence of
  - Alignment padd

A solution: **dynamic analysis**

- run the program multiple times, and observe the targets of indirect jumps and calls
- use this info to expand your graphs

Dynamic Analysis

- Program analysis technique performed at runtime
- Historically, has been used for performance monitoring and software testing.
- Security related behavioral analysis is all based on dynamic analysis. e.g. malware analysis
- Security testing – fuzzing.
- Can complement static analysis by providing missing info.



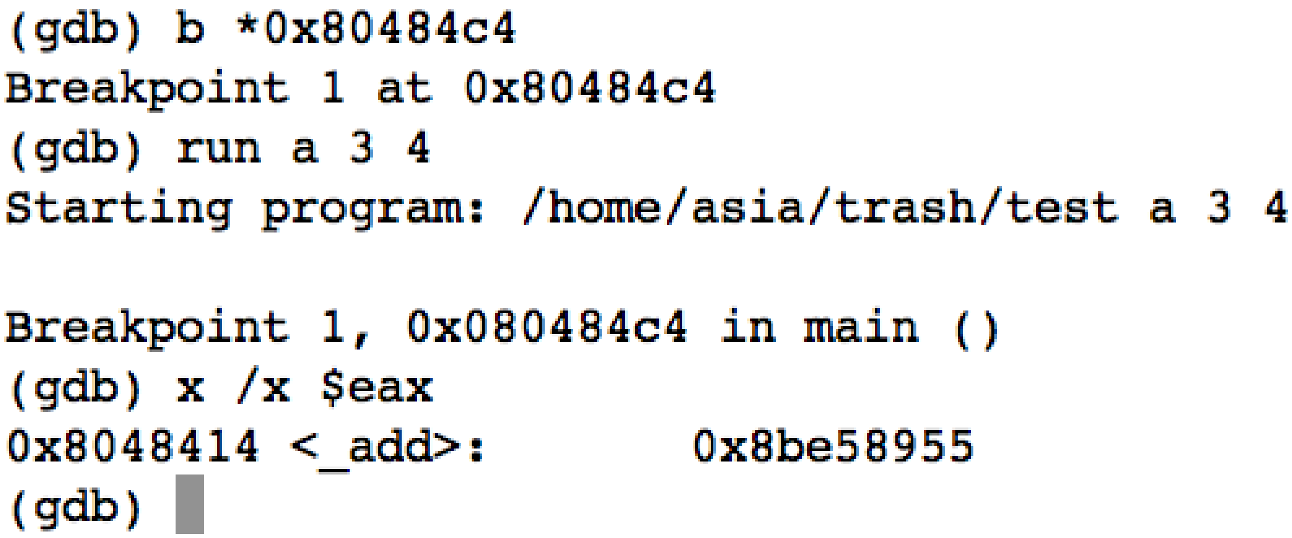
```
int _add(int arg1, int arg2) {
        return arg1 + arg2;
}


int _div(int arg1, int arg2) {
        if (arg2 == 0) return 0;
        return arg1/arg2;
}


int main(int argc, char **argv)
{
        int (*fun)(int, int);
        int result = 0;

        char* c = argv[1];
        int arg1 = atoi(argv[2]);
        int arg2 = atoi(argv[3]);

        fun = 0;
        if (c[0] == 'a') {
                fun = _add;
        } else if (c[0] == 'd') {
                fun = _div;
        }

        result =  fun(arg1, arg2);
        printf("%s(%d, %d)=%d\n",
                (c[0]== 'a')?"add":((c[1]=='d')?"div":""),
                arg1, arg2, result);

        return 0;
}
~
~
```

```
8048478:        call    8048350 <atoi@plt>
804847d:        mov     %eax,0x3c(%esp)
8048481:        movl    $0x0,0x2c(%esp)
8048488:
8048489:        mov     0x34(%esp),%eax
804848d:        movzbl  (%eax),%eax
8048490:        cmp     $0x61,%al
8048492:        jne     804849e <main+0x60>
8048494:        movl    $0x8048414,0x2c(%esp)
804849b:
804849c:        jmp     80484b1 <main+0x73>
804849e:        mov     0x34(%esp),%eax
80484a2:        movzbl  (%eax),%eax
80484a5:        cmp     $0x64,%al
80484a7:        jne     80484b1 <main+0x73>
80484a9:        movl    $0x8048421,0x2c(%esp)
80484b0:
80484b1:        mov     0x3c(%esp),%eax
80484b5:        mov     %eax,0x4(%esp)
80484b9:        mov     0x38(%esp),%eax
80484bd:        mov     %eax,(%esp)
80484c0:        mov     0x2c(%esp),%eax
80484c4:        call    *%eax
80484c6:        mov     %eax,0x30(%esp)
80484ca:        mov     0x34(%esp),%eax
80484ce:        movzbl  (%eax),%eax
80484d1:        cmp     $0x61,%al
80484d3:        je      80484f1 <main+0xb3>
80484d5:        mov     0x34(%esp),%eax
```

How shall we monitor `call` instructions at run-time?

How shall we monitor `call` instructions at run-time?

- Use gdb



```
(gdb) b *0x80484c4
Breakpoint 1 at 0x80484c4
(gdb) run a 3 4
Starting program: /home/asia/trash/test a 3 4

Breakpoint 1, 0x080484c4 in main ()
(gdb) x /x $eax
0x8048414 <_add>:       0x8be58955
(gdb) 
```

How shall we monitor `call` instructions at run-time?

- Use `gdb` to break on an instruction
  - Lots of manual effort!

How shall we monitor `call` instructions at run-time?

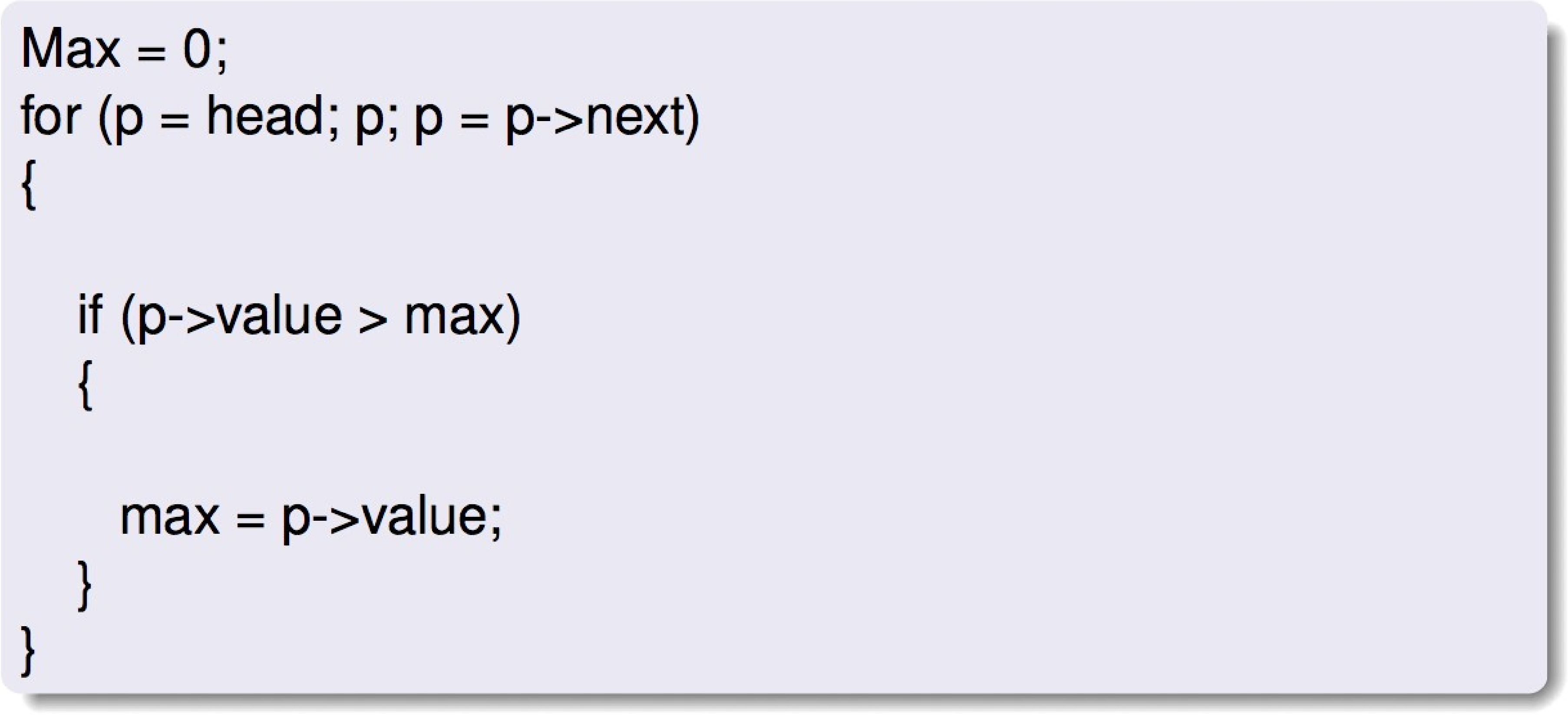
- Use `gdb` to break on an instruction
  - Lots of manual effort!
- **Instrument** the binary to log the targets of all indirect `call`/`jump` instructions for us
  - Automatically!
  -

Instrumentation

A technique that injects instrumentation code into a binary to collect run-time information

Instrumentation

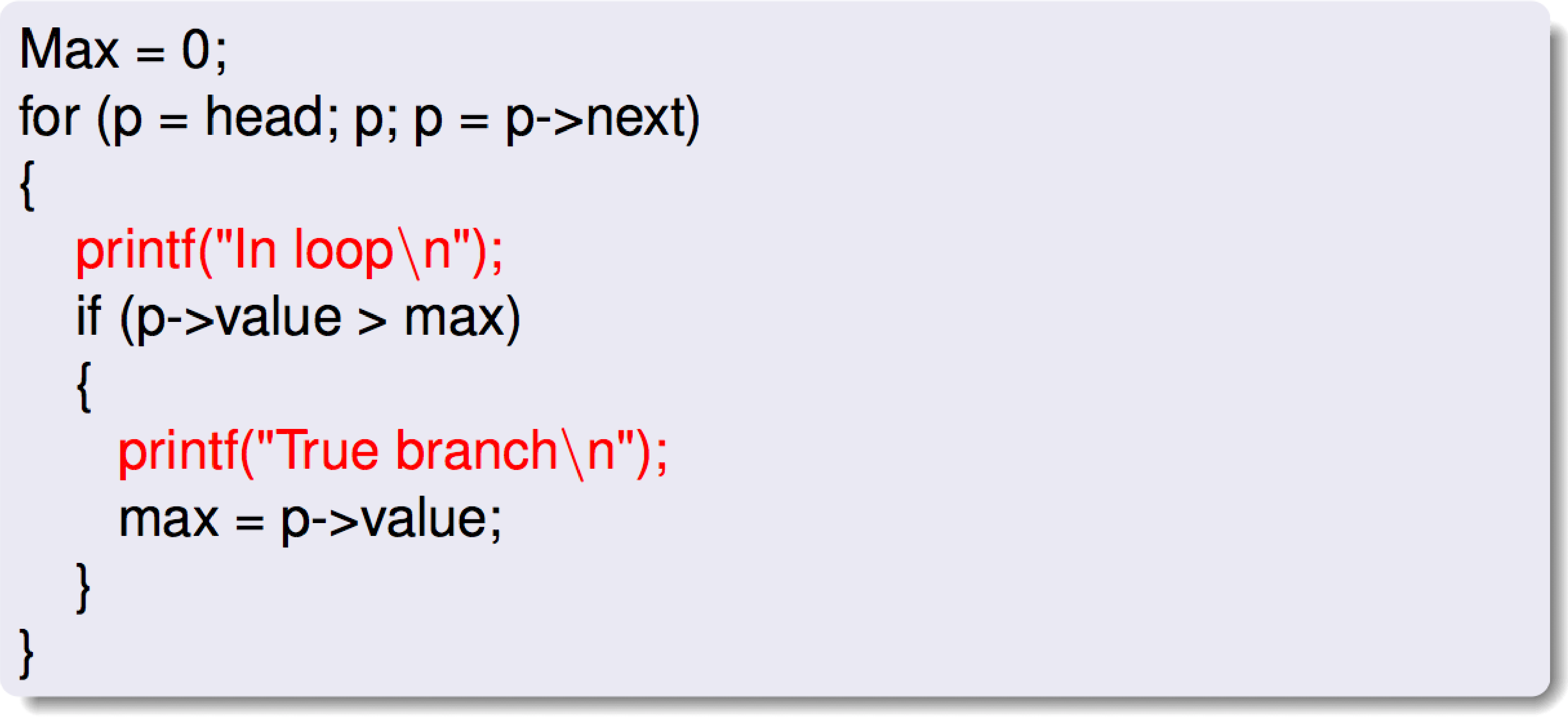
A technique that injects instrumentation code into a binary to collect run-time information



```
Max = 0;
for (p = head; p; p = p->next)
{

   if (p->value > max)
   {

     max = p->value;
   }
}
```

Instrumentation

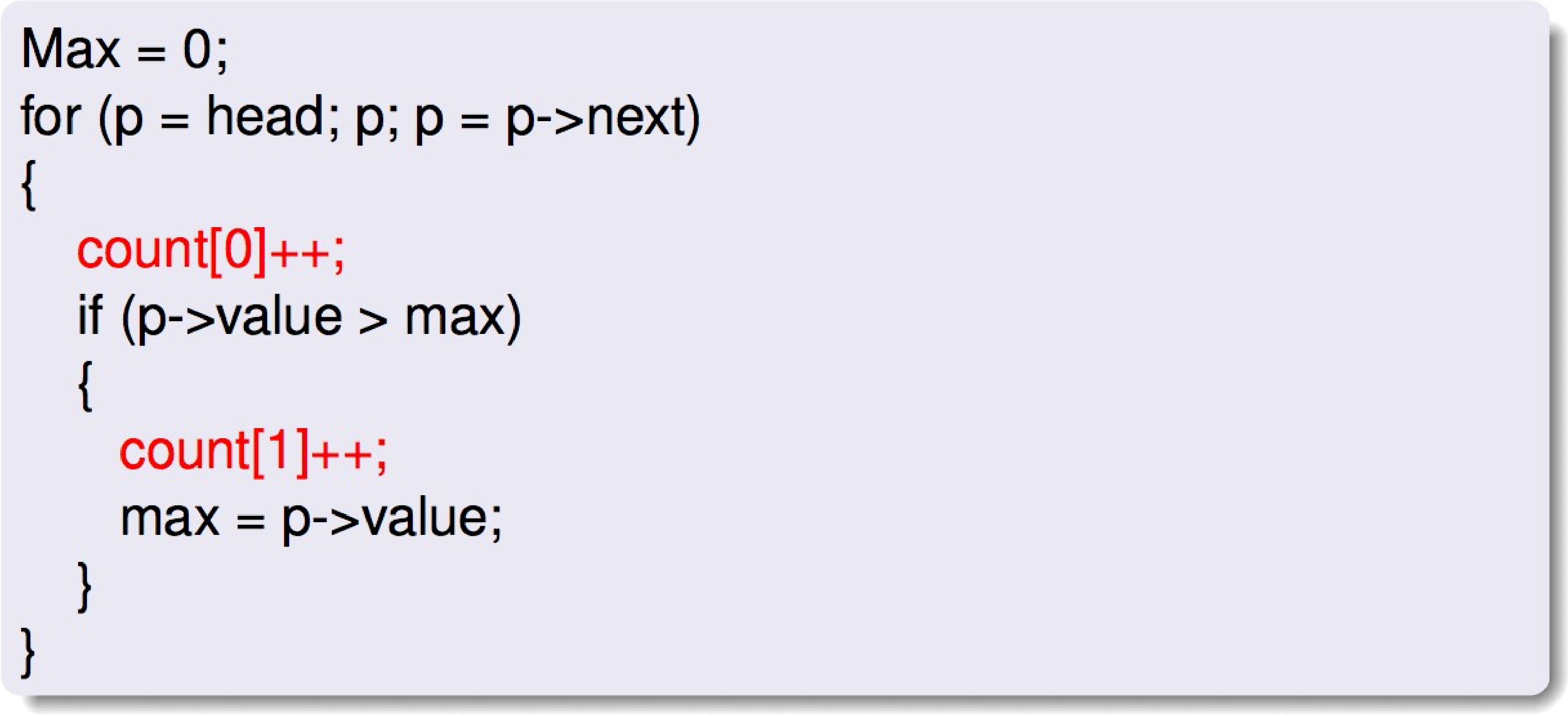
A technique that injects instrumentation code into a binary to collect run-time information



```
Max = 0;
for (p = head; p; p = p->next)
{
    printf("In loop\n");
    if (p->value > max)
    {
        printf("True branch\n");
        max = p->value;
    }
}
```

Instrumentation

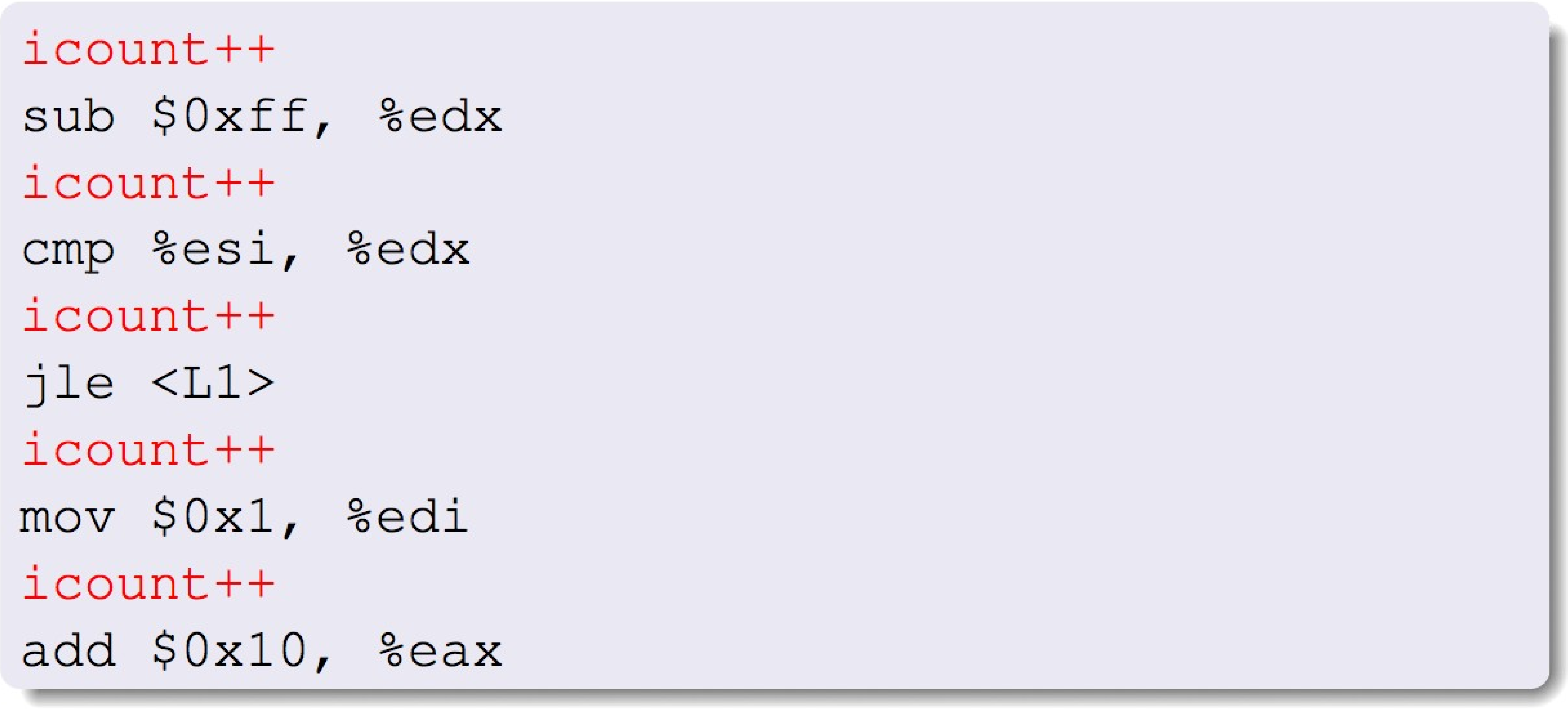
A technique that injects instrumentation code into a binary to collect run-time information



```
Max = 0;
for (p = head; p; p = p->next)
{
   count[0]++;
   if (p->value > max)
   {
     count[1]++;
     max = p->value;
   }
}
```

Instrumentation

A technique that injects instrumentation code into a binary to collect run-time information



```
icount++
sub $0xff, %edx
icount++
cmp %esi, %edx
icount++
jle <L1>
icount++
mov $0x1, %edi
icount++
add $0x10, %eax
```

Instrumentation

A technique that injects instrumentation code into a binary to collect run-time information

Instrumentation

A technique that injects instrumentation code into a binary to collect run-time information

- It executes as a part of the normal instruction stream

Instrumentation

A technique that injects instrumentation code into a binary to collect run-time information

- It executes as a part of the normal instruction stream
- It doesn’t modify the semantics of the program

When is instrumentation useful?

When is instrumentation useful?

- Profiling for compiler optimization/performance profiling:
  - Instruction profiling
  - Basic block count
  - Value profile

When is instrumentation useful?

- Profiling for compiler optimization/performance profiling:
  - Instruction profiling
  - Basic block count
  - Value profile
- Bug detection/Vulnerability identification/Exploit generation:
  - Find references to uninitialized, unallocated addresses
  - Inspect arguments at a particular function call
  - Inspect function pointers and return addresses
  - Record & replay

When is instrumentation useful?

- Profiling for compiler optimization/performance profiling:
  - Instruction profiling
  - Basic block count
  - Value profile
- Bug detection/Vulnerability identification/Exploit generation:
  - Find references to uninitialized, unallocated addresses
  - Inspect arguments at a particular function call
  - Inspect function pointers and return addresses
  - Record & replay
- Architectural research: processor and cache simulation, trace collection

Instrumentation

Instrumentation

- **Static instrumentation** – instrument before runtime
  - Source code instrumentation
    - Instrument source programs (e.g., clang’s source-to-source transformation)
  - IR instrumentation
    - Instrument compiler-generated IR (e.g., LLVM)
  - Binary instrumentation
    - Instrument executables directly by inserting additional assembly instructions (e.g., Dyninst)

Instrumentation

- **Static instrumentation** – instrument before runtime
  - Source code instrumentation
    - Instrument source programs (e.g., clang's source-to-source transformation)
  - IR instrumentation
    - Instrument compiler-generated IR (e.g., LLVM)
  - Binary instrumentation
    - Instrument executables directly by inserting additional assembly instructions (e.g., Dyninst)

- **Dynamic binary instrumentation** – instrument at runtime
  - Instrument code just before it runs (Just in time – JIT)
  - E.g., Pin, Valgrind, DynamoRIO, QEMU

Why **binary** instrumentation

Why **binary** instrumentation

- Libraries are a big pain for source/IR-level instrumentation
  - Proprietary libraries: communication (MPI, PVM), linear algebra (NGA), database query (SQL libraries)

Why **binary** instrumentation

- Libraries are a big pain for source/IR-level instrumentation
  - Proprietary libraries: communication (MPI, PVM), linear algebra (NGA), database query (SQL libraries)
- Easily handles multi-lingual programs
  - Source code level instrumentation is heavily language dependent.

Why **binary** instrumentation

- Libraries are a big pain for source/IR-level instrumentation
  - Proprietary libraries: communication (MPI, PVM), linear algebra (NGA), database query (SQL libraries)
- Easily handles multi-lingual programs
  - Source code level instrumentation is heavily language dependent.
- Worms and viruses are rarely provided with source code

Why **binary** instrumentation

- Libraries are a big pain for source/IR-level instrumentation
  - Proprietary libraries: communication (MPI, PVM), linear algebra (NGA), database query (SQL libraries)
- Easily handles multi-lingual programs
  - Source code level instrumentation is heavily language dependent.
- Worms and viruses are rarely provided with source code
- And WYSINWYX!

Dynamic binary instrumentation

- **Pros**
  - No need to recompile or relink
  - Discovers code at runtime
  - Handles dynamically generated code
  - Attaches to running processes (some tools)
- **Cons**
  - Usually higher performance overhead
  - Requires a framework which can be detected by malware

Pin

A Dynamic Binary Instrumentation Tool

Pin

A Dynamic Binary Instrumentation Tool

1. What can we do with Pin?
2. How does it work?
3. Examples (original Pin examples)
4. Performance overhead

Pin

- Pin is a tool for the instrumentation of programs. It supports Linux* and Windows* executables for x86, x86_64, and IA-64 architectures.
- Pin allows a tool to insert arbitrary code (written in C or C++) in arbitrary places in the executable. The code is added dynamically while the executable is running. This also makes it possible to attach Pin to an already running process.

What can we do with Pin?

What can we do with Pin?

- Fully examine any (type of) x86 instruction
  - Insert a call to your own function which gets called when that instruction executes
    - Parameters: register values (including IP), memory addresses, memory contents…

What can we do with Pin?

- Fully examine any (type of) x86 instruction
  - Insert a call to your own function which gets called when that instruction executes
    - Parameters: register values (including IP), memory addresses, memory contents…
- Track function calls, including library calls and syscalls
  - Examine/change arguments
  - Insert function hooks: replace application/library functions with your own

What can we do with Pin?

- Fully examine any (type of) x86 instruction
  - Insert a call to your own function which gets called when that instruction executes
    - Parameters: register values (including IP), memory addresses, memory contents…
- Track function calls, including library calls and syscalls
  - Examine/change arguments
  - Insert function hooks: replace application/library functions with your own
- Track application threads

What can we do with Pin?

- Fully examine any (type of) x86 instruction
  - Insert a call to your own function which gets called when that instruction executes
    - Parameters: register values (including IP), memory addresses, memory contents…
- Track function calls, including library calls and syscalls
  - Examine/change arguments
  - Insert function hooks: replace application/library functions with your own
- Track application threads
- And more ⏪

What can we do with Pin?

- Fully examine any (type of) x86 instruction
  - Insert a call to your own function which gets called when that instruction executes
    - Parameters: register values (including IP), memory addresses, memory contents…
- Track function calls, including library calls and syscalls
  - Examine/change arguments
  - Insert function hooks: replace application/library functions with your own
- Track application threads
- And more ◀◀

If Pin doesn't have it, you don't want it ;)

Advantages of Pin

- **Easy-to-use Instrumentation:**
  - Uses dynamic instrumentation
    - Does not need source code, recompilation, post-linking
- **Programmable Instrumentation:**
  - Provides rich APIs to write in C/C++ your own instrumentation tools (called Pintools)
- **Multiplatform**:
  - Supports x86, x86_64
  - Supports Linux, Windows binaries
- **Robust**:
  - Instruments real-life applications: Database, web browsers,. . .
  - Instruments multithreaded applications
  - Supports signals
- **Efficient**:
  - Applies compiler optimizations on instrumentation code

Usage of Pin at Intel

- Profiling and analysis products
  - Intel Parallel Studio
    - Amplifier (Performance Analysis)
      - Lock and waits analysis
      - Concurrency analysis
    - Inspector (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

Pin usage outside Intel

Pin usage outside Intel

- **Popular and well supported**
  - 100,000+ downloads,
  - 4226+ citations
  - (as of 2019-2020)

- **Free DownLoad**

https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool

  - **Includes: Detailed user manual, source code for 100s of Pin tools**

- **Pin User Group (PinHeads)**
  - **https://groups.io/g/pinheads**
  - **Pin users and Pin developers answer questions**

Pin usage outside Intel

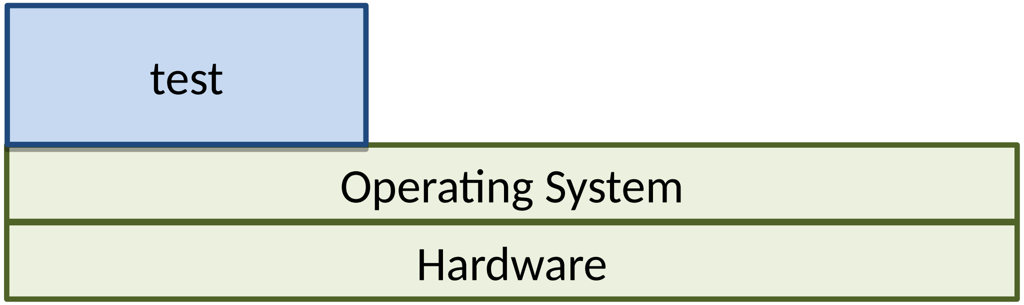
- **Popular and well supported**
  - 100,000+ downloads,
  - 4226+ citations
  - (as of 2019-2020)
- **Free DownLoad**

https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool

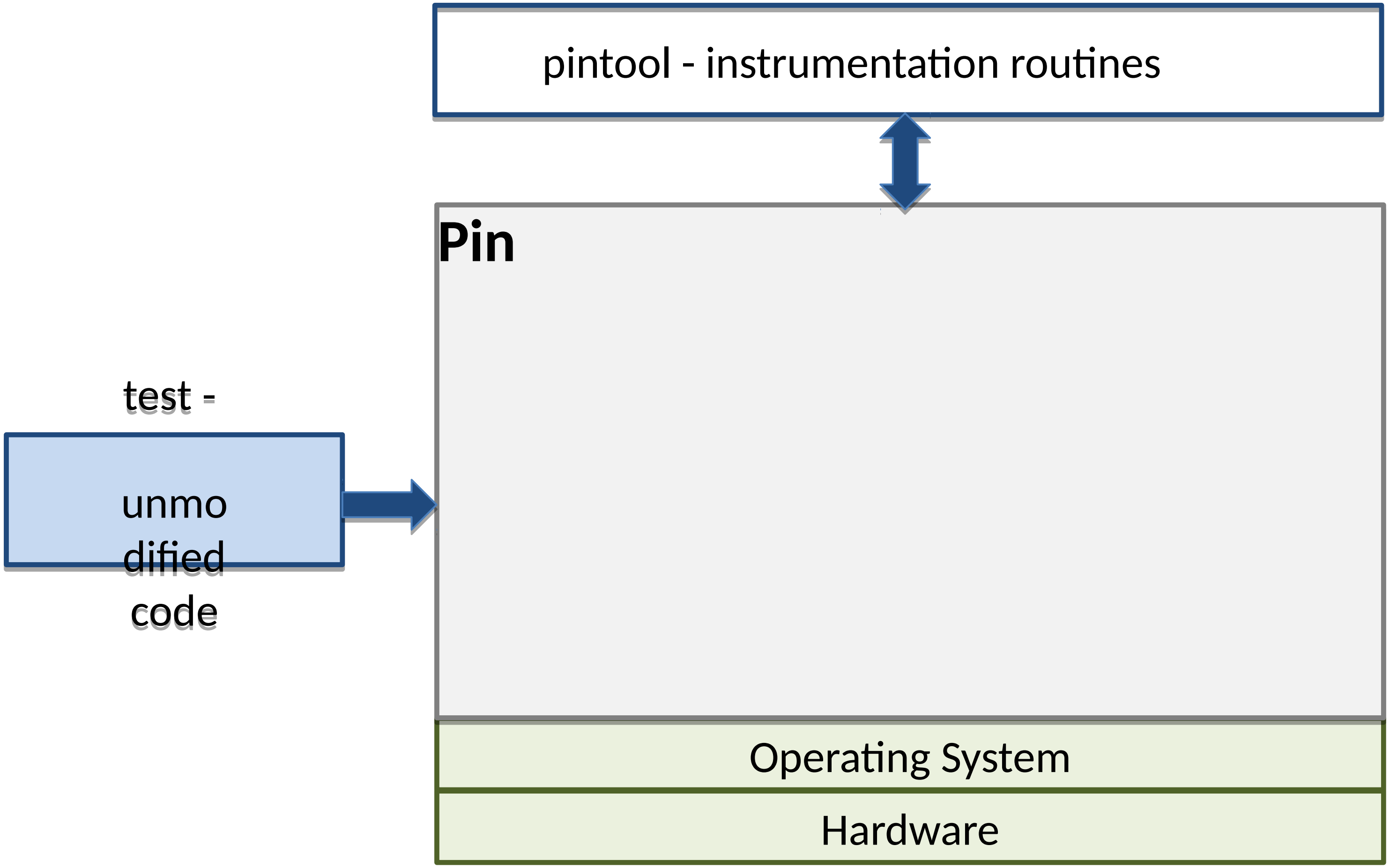
  - **Includes: Detailed user manual, source code for 100s of Pin tools**
- **Pin User Group (PinHeads)**
  - **https://groups.io/g/pinheads**
  - **Pin users and Pin developers answer questions**

Architecture overview

./test



./pin –t pintool -- test



pintool - instrumentation routines

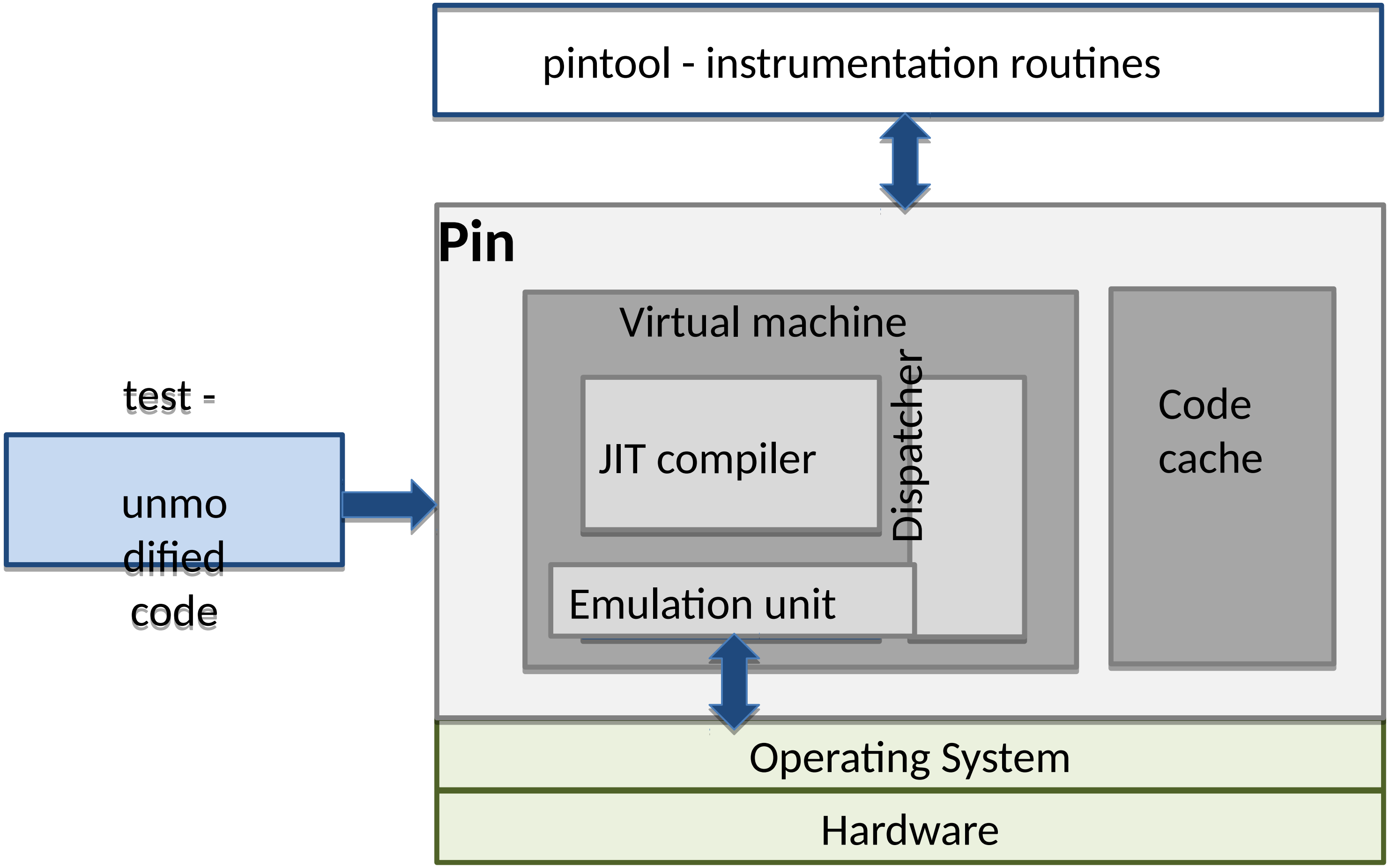
Pin

test - unmodified code

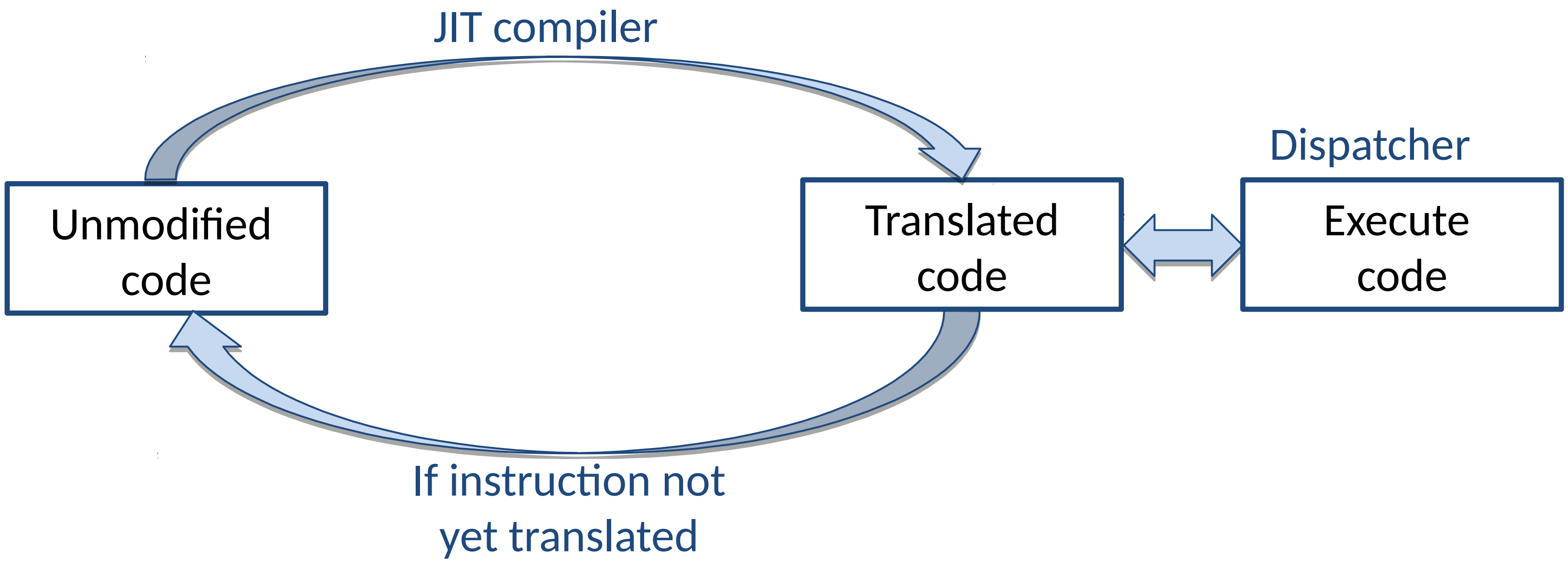
Operating System

Hardware

./pin –t pintool -- test



JIT compilation



JIT compiler

Unmodified code

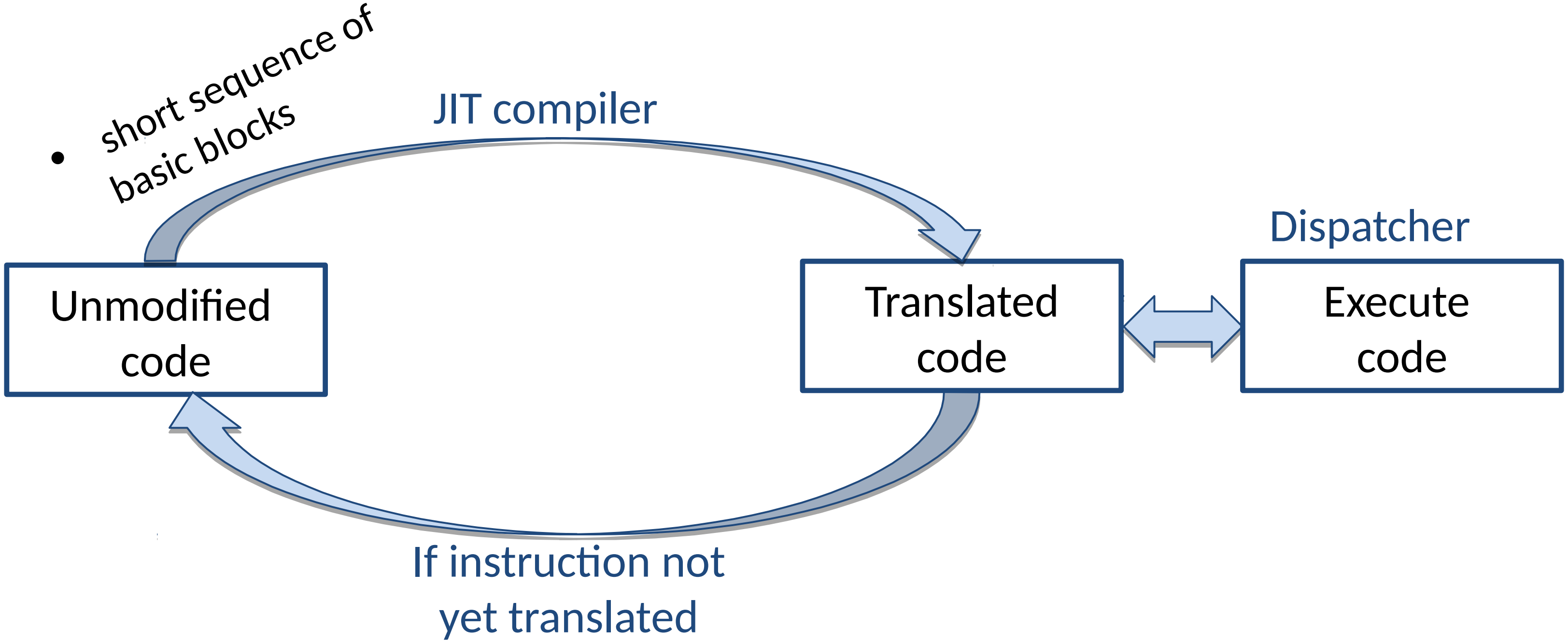
Translated code

Dispatcher

Execute code

If instruction not yet translated

JIT compilation



- short sequence of basic blocks

JIT compiler

Dispatcher

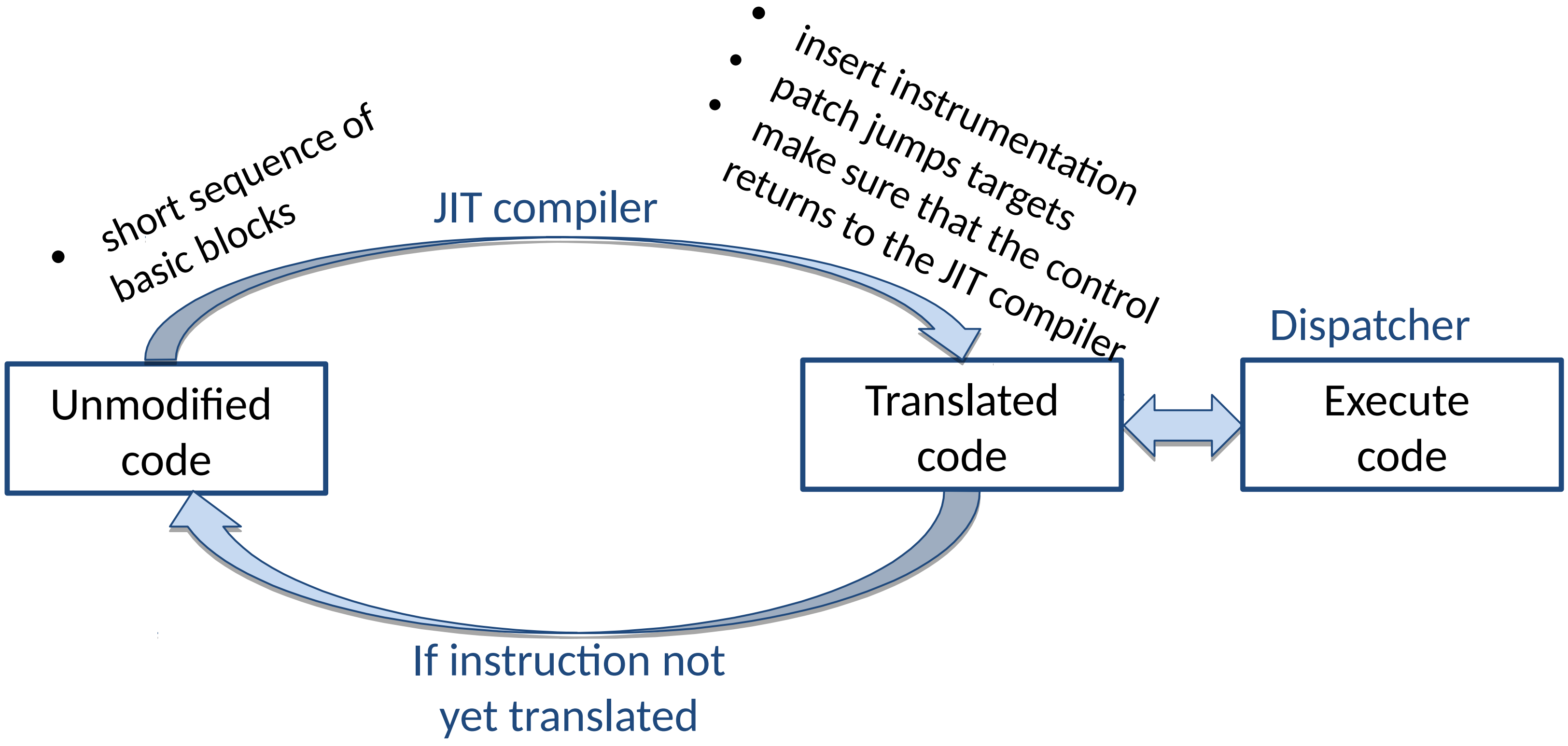
Unmodified code

Translated code

Execute code

If instruction not yet translated

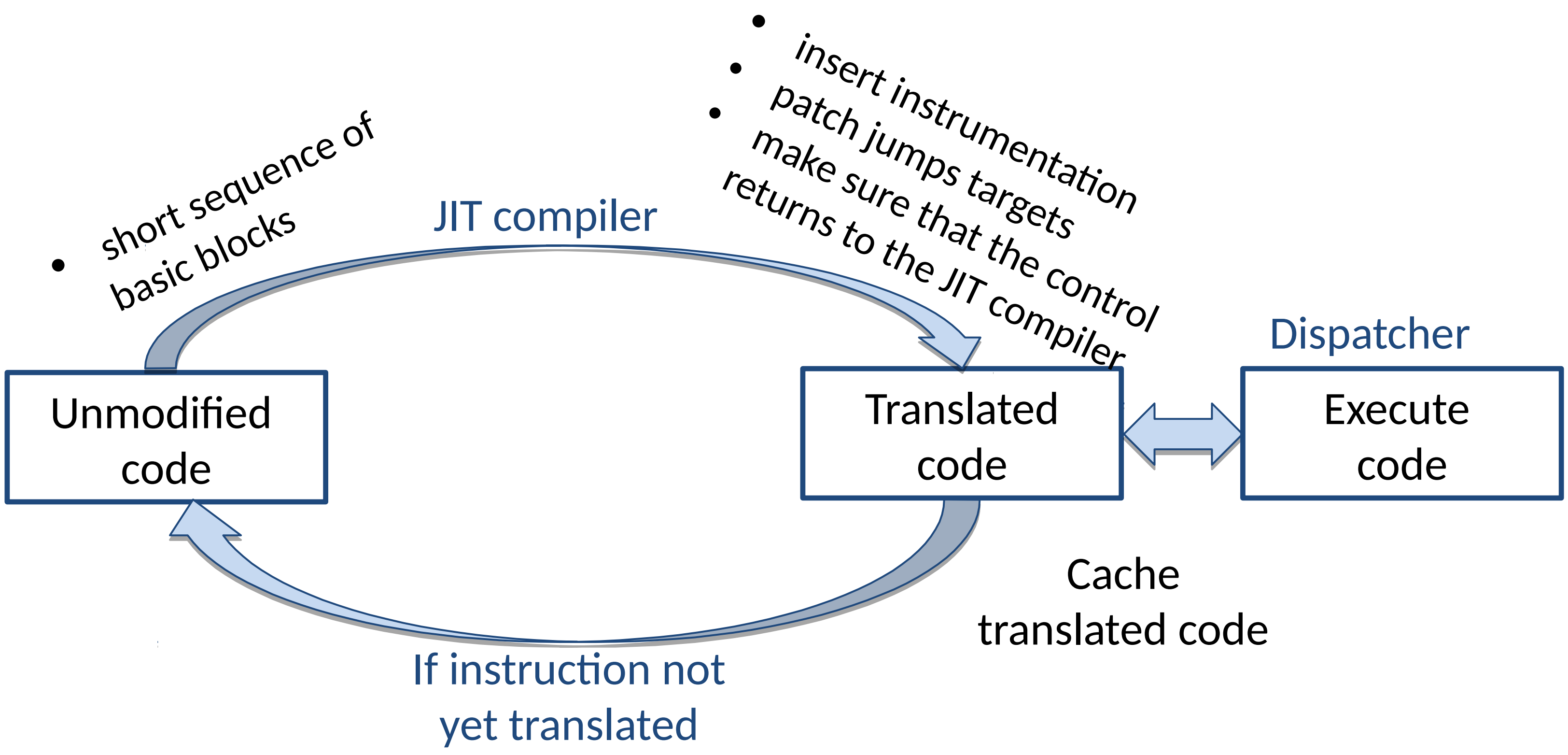
JIT compilation



- short sequence of basic blocks

- insert instrumentation
- patch jumps targets
- make sure that the control returns to the JIT compiler

JIT compila



- short sequence of basic blocks

JIT compiler

- insert instrumentation
- patch jumps targets
- make sure that the control returns to the JIT compiler

Dispatcher

Unmodified code

Translated code

Execute code

Cache translated code

If instruction not yet translated

Example 1: docount
- instruction counting tool

Instruction counting tool

```
#include “pin.h”
uint64_t 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;
}
```

Instruction counting tool

```
#include “pin.h”
uint64_t 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;
}
```

Initialize PIN

Instruction counting tool

```
#include “pin.h”
uint64_t 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;
}
```

INS is valid only inside this routine.

Instrumentation routine; called during jitting of INS.

Register instruction instrumentation routine

Instruction counting tool

```
#include “pin.h”
uint64_t 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;
}
```

Analysis routine; Executes each time jitted INStruction executes.

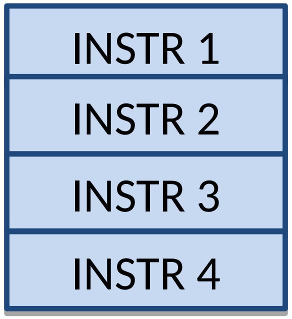
Instruction counting tool

```
#include “pin.h”
uint64_t 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;
}
```

Question: which function gets executed more often?

Instruction counting tool

Native BB



INSTR 1

INSTR 2

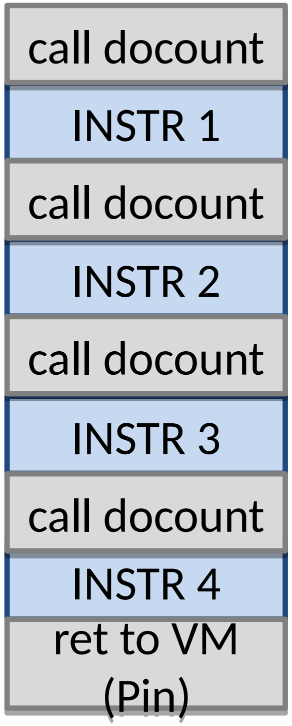
INSTR 3

INSTR 4

Analysis routine (`docount`)

Translation time (JIT compilation)

Rewriting only! We don’t execute the native BB now!



call docount

INSTR 1

call docount

INSTR 2

call docount

INSTR 3

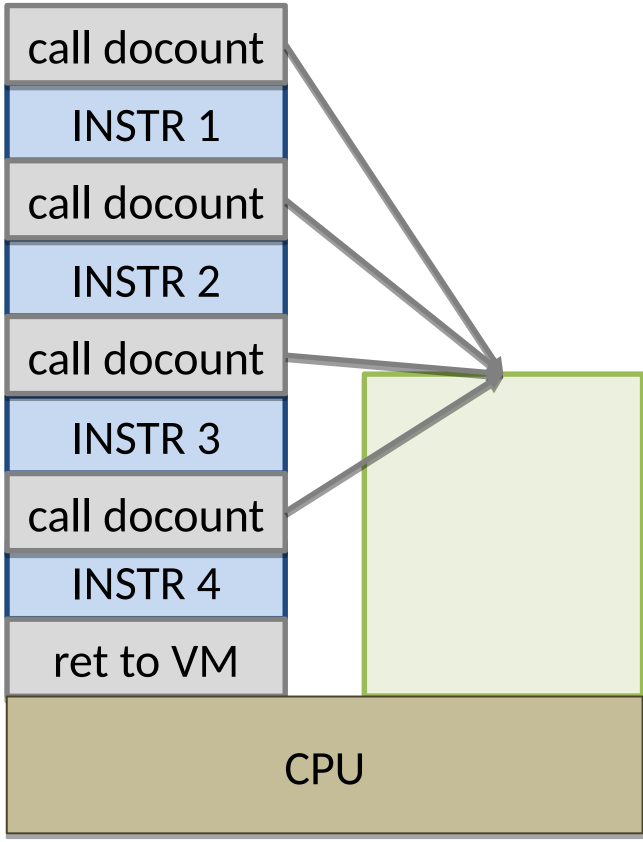
call docount

INSTR 4

ret to VM (Pin)

Call the `Instruction` **instrumentation routine** to see that we need to **insert a call** to `docount` before every instruction.
To be precise, `INSTR X` are not necessarily exactly the same, but very little is changed.

Execution time (Dispatcher)



call docount

INSTR 1

call docount

INSTR 2

call docount

INSTR 3

call docount

INSTR 4

ret to VM

CPU

Execute the translated block. During the execution, the **analysis routines are executed**. In our case - `docount`.

Instrumentation vs Analysis

- **Instrumentation routines**
  - Define where instrumentation is inserted, e.g., before instruction
  - Invoked when **an instruction is being jitted**
- **Analysis routines**
  - Define what to do when instrumentation is activated, e.g., increment counter
  - Invoked every time **an instruction is executed**

Instruction counting tool

```
#include “pin.h”
uint64_t 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;
}
```

Instruction counting tool

```
#include “pin.h”
uint64_t 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;
}
```

- **`sub $0xff, %edx`**

Instruction counting tool

```
#include “pin.h”
uint64_t 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;
}
```

```
switch to pin stack
save registers
call docount
restore registers
switch to app stack
```

- `sub $0xff, %edx`

Instruction counting tool

```
#include “pin.h”
uint64_t 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;
}
```

- **sub $0xff, %edx**

Instruction counting tool

```
#include “pin.h”
uint64_t 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;
}
```

- **`sub $0xff, %edx`**
- **`cmp %esi,  %edx`**

Instruction counting tool

```
#include “pin.h”
uint64_t 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;
}
```

- `sub $0xff,  %edx`
  `inc icount`
- `cmp %esi,   %edx`

Instruction counting tool

```
#include “pin.h”
uint64_t 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;
}
```

- **sub $0xff, %edx**
  inc icount
- **cmp %esi, %edx**
- **jle   <L1>**

Instruction counting tool

```
#include “pin.h”
uint64_t 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;
}
```

- **sub $0xff, %edx**

  inc icount
- **cmp %esi, %edx**

  inc icount
- **jle <L1>**

Instruction counting tool

```
#include “pin.h”
uint64_t 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;
}
```

- **sub $0xff, %edx**
  inc icount
- **cmp %esi, %edx**
  save eflags
  inc icount
  restore eflags
- **jle   <L1>**

Instruction counting tool

```
#include “pin.h”
uint64_t 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;
}
```

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

Instruction counting tool

```
#include “pin.h”
uint64_t 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;
}
```

- **sub $0xff, %edx**

  inc icount
- **cmp %esi, %edx**

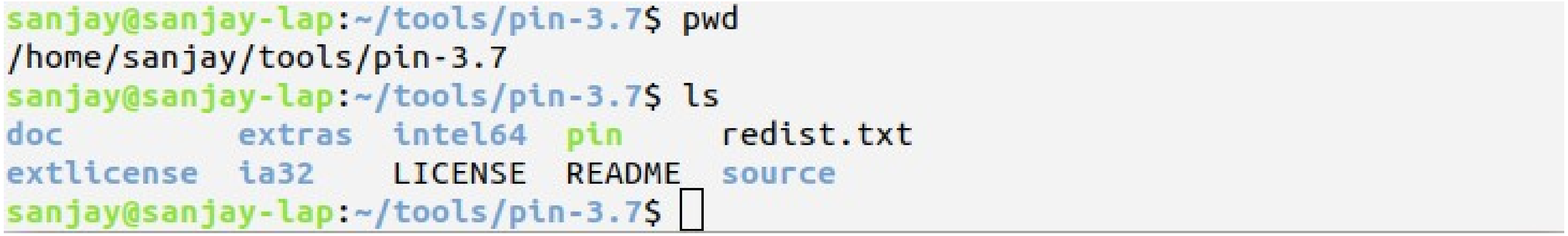
  save eflags
  inc icount
  restore eflags
- **jle <L1>**

  inc icount
- **mov 0x1, %edi**

Pin execution

Pin execution

1. Download Pin from http://www.pintool.org



```
sanjay@sanjay-lap:~/tools/pin-3.7$ pwd
/home/sanjay/tools/pin-3.7
sanjay@sanjay-lap:~/tools/pin-3.7$ ls
doc         extras  intel64  pin     redist.txt
extlicense  ia32    LICENSE  README  source
sanjay@sanjay-lap:~/tools/pin-3.7$ 
```

Pin execution

2. Write your own pintool.

- Numerous examples:

```
sanjay@sanjay-lap:~/tools/pin-3.7/source/tools$ ls
```

- Our instruction counting tool

```
make obj-intel64/inscount0.so TARGET=intel64
```

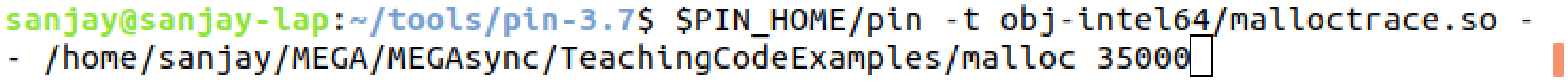
Pin execution

3. Set the `PIN_HOME` environment variable to your Pin directory, and `make`.

```
sanjay@sanjay-lap:~/tools/pin-3.7$ export
PIN_HOME=~/tools/pin-3.7/
```

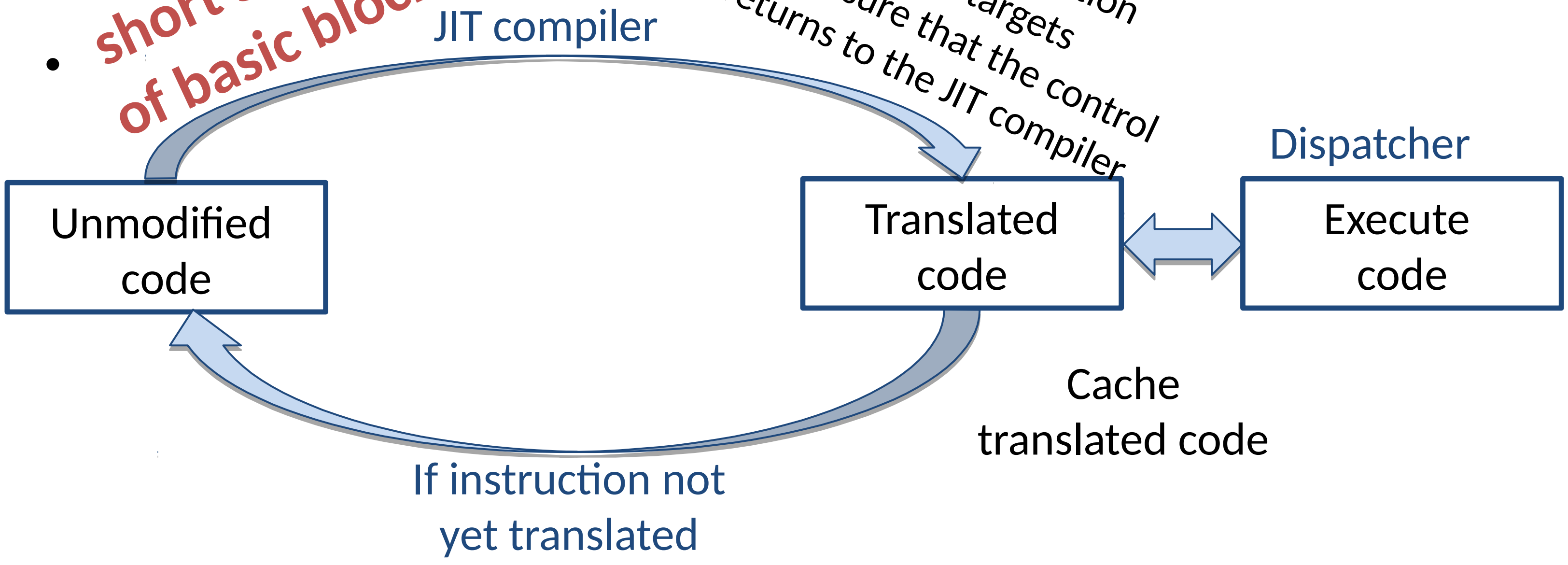
Pin execution

4. Run ⏪



```
sanjay@sanjay-lap:~/tools/pin-3.7$ $PIN_HOME/pin -t obj-intel64/malloctrace.so -
- /home/sanjay/MEGA/MEGAsync/TeachingCodeExamples/malloc 35000
```

JIT compila



- short sequence of basic blocks

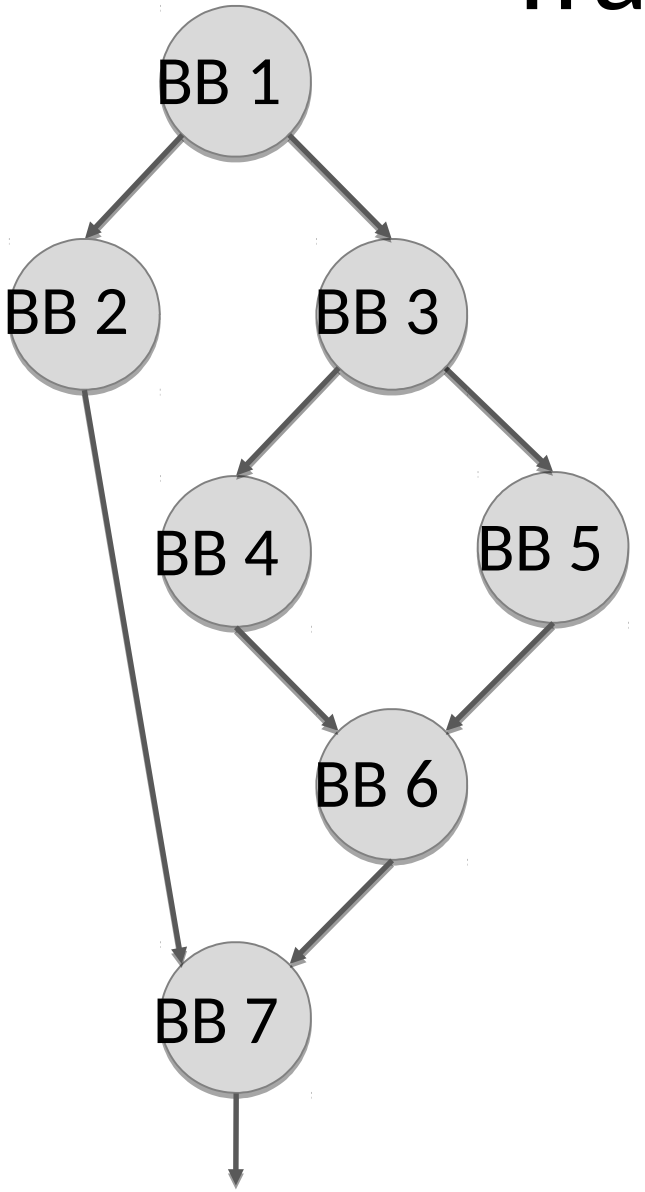
- insert instrumentation
- patch jumps targets
- make sure that the control returns to the JIT compiler

Trace

- The application is compiled one **trace** at a time
  - A sequence of basic blocks with one entry point
  - It terminates at one of the conditions:
    - an unconditional control transfer, i.e., jmp/call/ret
    - a pre-defined number of conditional control transfers
    - a pre-defined number of instructions
  - Always ends with a stub which redirects control back to the VM

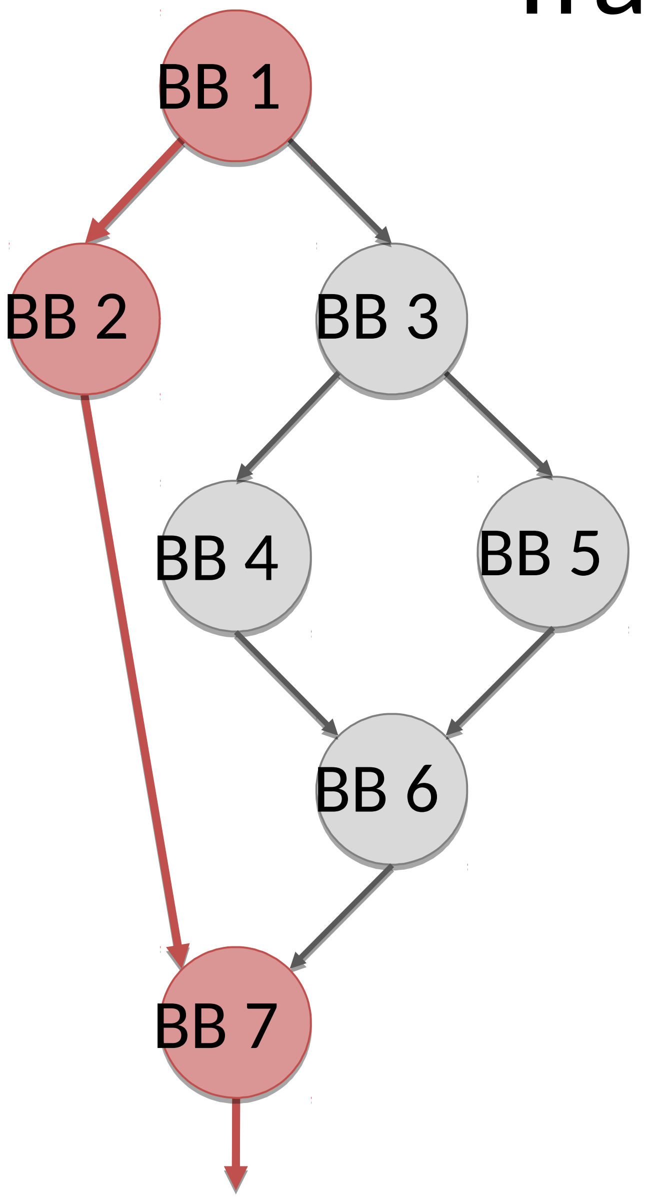
Trace

Original code



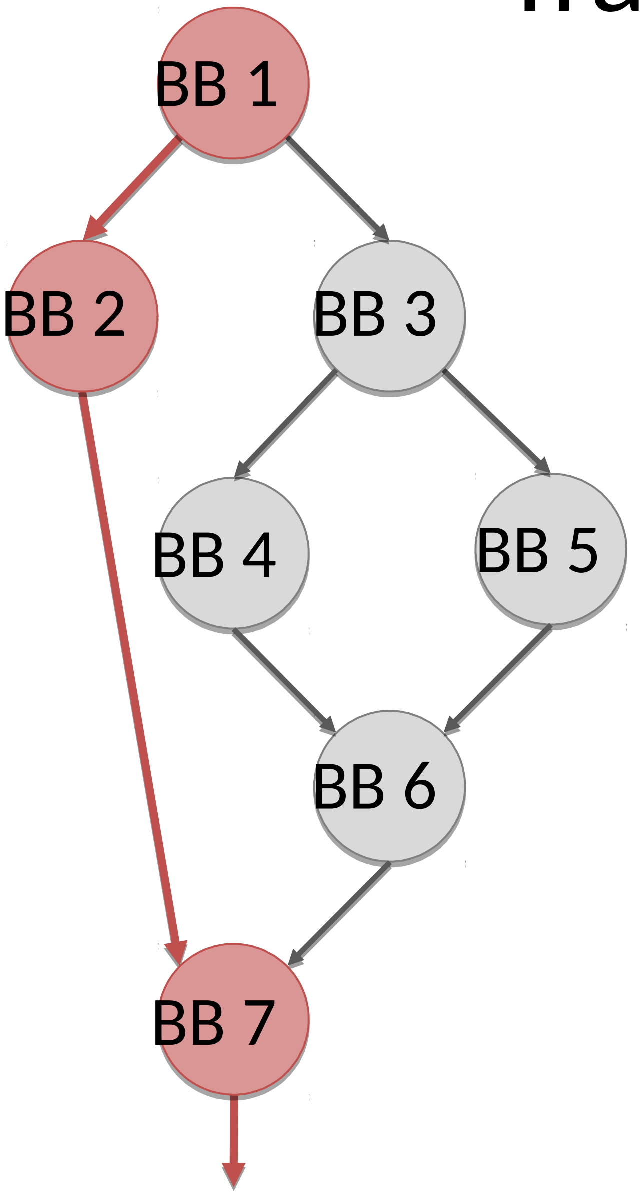
Trace

Original code

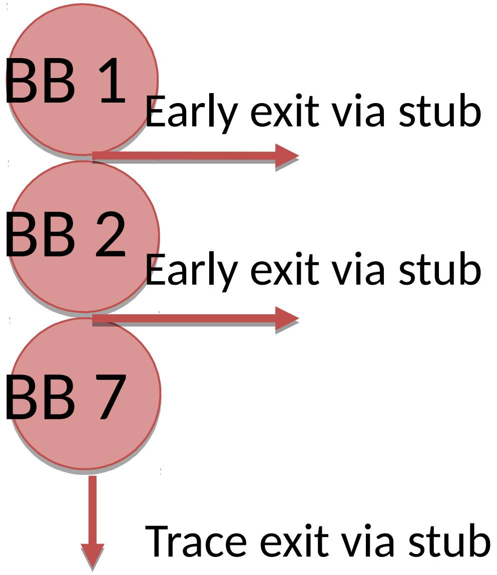


Trace

Original code



Translated trace



Early exit via stub

Early exit via stub

Trace exit via stub

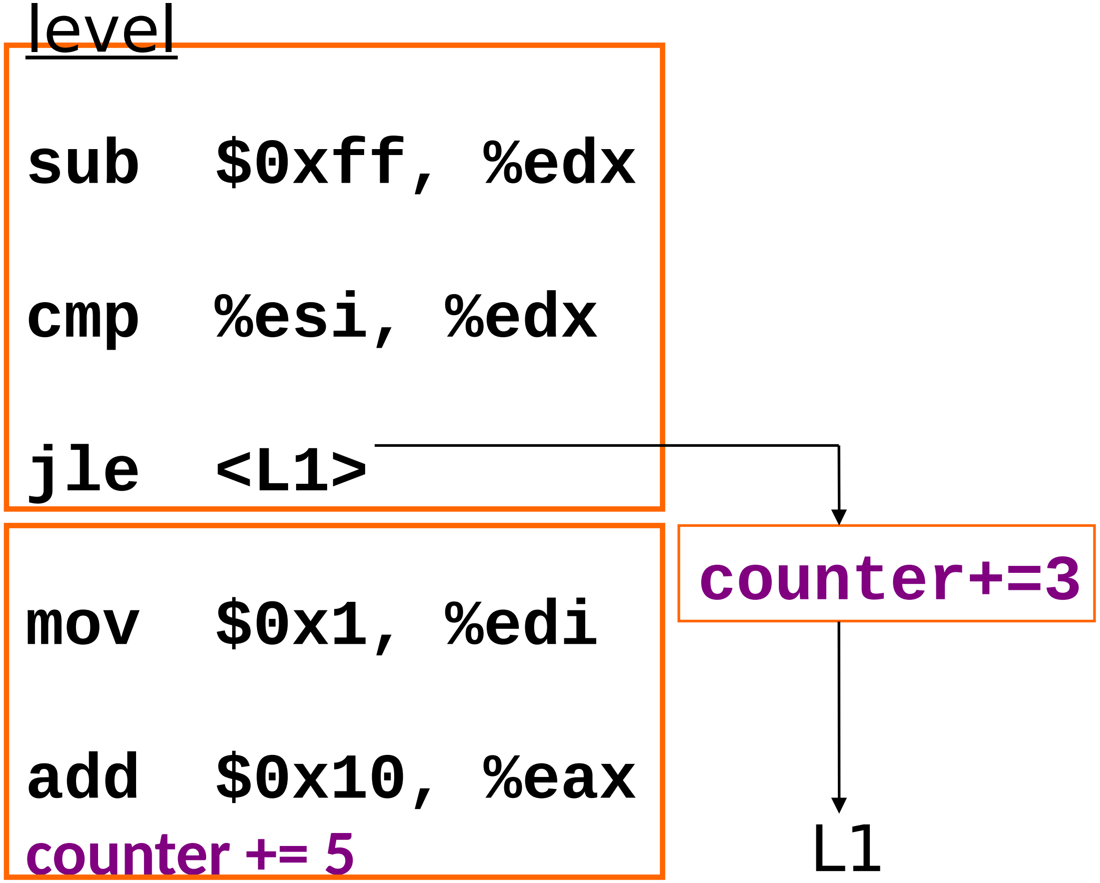
Counting at the BBL/Trace level

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
```

Example 2: docount++
- instruction counting tool optimized

Instruction counting tool

```
#include “pin.h”
uint64_t 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;
}
```

Instruction counting tool++

```
#include “pin.h”
uint64_t icount = 0;

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

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

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

int main(int argc, char **argv) {
   PIN_Init(argc, argv);
   TRACE_AddInstrumentFunction(Trace, 0);
   PIN_AddFiniFunction(Fini, 0);
   PIN_StartProgram();  // never returns
   return 0;
}
```

Direct Pin to call the pintool `Trace` function at the beginning of jitting of each trace.

Instruction countir

A handle to the currently jitted trace.

Use it to iterate through the BBLs of this trace.

```
#include “pin.h”
uint64_t icount = 0;

void PIN_FAST_ANALYSIS_CALL docount(INT32

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

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

int main(int argc, char **argv) {
   PIN_Init(argc, argv);
   TRACE_AddInstrumentFunction(Trace, 0);
   PIN_AddFiniFunction(Fini, 0);
   PIN_StartProgram();  // never returns
   return 0;
}
```

Instruction counting tool++

```
#include “pin.h”
uint64_t icount = 0;

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

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

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

int main(int argc, char **argv) {
   PIN_Init(argc, argv);
   TRACE_AddInstrumentFunction(Trace, 0);
   PIN_AddFiniFunction(Fini, 0);
   PIN_StartProgram();  // never returns
   return 0;
}
```

Call **docount** before executing each BBL.

Pass an arg of type `IARG_UINT32`, and value `BBL_NumIns(bbl)`.

Instruction counting tool++

```
#include “pin.h”
uint64_t icount = 0;

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

void Trace(TRACE trace, void *v) {
   for(BBL bbl=TRACE_BBlHead(trace);
           BBL_Valid(bbl); bbl=BBL_Next(bbl))
           BBL_InsertCall(ins, IPOINT_ANYWHERE,
               (AFUNPTR) docount, IARG_FAST_ANALYSIS_C
               IARG_UINT32, BBL_NumIns(bbl), IA
}

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

int main(int argc, char **argv) {
   PIN_Init(argc, argv);
   TRACE_AddInstrumentFunction(Trace, 0);
   PIN_AddFiniFunction(Fini, 0);
   PIN_StartProgram();  // never returns
   return 0;
}
```

Insert the instrumentation anywhere in the BBL – this might enable Pin find an optimal place .

Example 3: Memory read logger

Memory read logger tool

```
#include “pin.h”
std::map<ADDRINT, std::string> disAssemblyMap;

void memoryRead(ADDRINT applicationIP,
   ADDRINT memoryReadAddress, UINT32 memoryReadSize) {
  printf(“0x%x %s reads %d bytes of memory at 0x%x\n”,
   applicationIP, disAssemblyMap[applicationIp].c_str(),
   memoryReadSize, memoryReadAddress);}

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)memoryRead,
           IARG_INST_PTR, // application IP
           IARG_MEMORYREAD_EA, // effective address of mem read
           IARG_MEMORY_READ_SIZE, IARG_END);
   }}

int main(int argc, char **argv) {
   PIN_Init(argc, argv);
   INS_AddInstrumentFunction(Instruction, 0);
   PIN_StartProgram();  // never returns
   return 0;
}
```

Memory read logger tool

```
#include “pin.h”
std::map<ADDRINT, std::string> disAssemblyMap;

void memoryRead(ADDRINT applicationIP,
   ADDRINT memoryReadAddress, UINT32 memoryReadSize) {
  printf(“0x%x %s reads %d bytes of memory at 0x%x\n”,
   applicationIP, disAssemblyMap[applicationIp].c_str(),
   memoryReadSize, memoryReadAddress);}

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)memoryRead,
           IARG_INST_PTR, // application IP
           IARG_MEMORYREAD_EA, // effective address of mem read
           IARG_MEMORY_READ_SIZE, IARG_END);
   }}

int main(int argc, char **argv) {
   PIN_Init(argc, argv);
   INS_AddInstrumentFunction(Instruction, 0);
   PIN_StartProgram();  // never returns
   return 0;
}
```

Instrumentation routine.

Memory read logger tool

```
#include “pin.h”
std::map<ADDRINT, std::string> disAssemblyMap;
```

Analysis routine.

```
void memoryRead(ADDRINT applicationIP,
   ADDRINT memoryReadAddress, UINT32 memoryReadSize) {
  printf(“0x%x %s reads %d bytes of memory at 0x%x\n”,
   applicationIP, disAssemblyMap[applicationIp].c_str(),
   memoryReadSize, memoryReadAddress);}
```

```
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)memoryRead,
           IARG_INST_PTR, // application IP
           IARG_MEMORYREAD_EA, // effective address of mem read
           IARG_MEMORY_READ_SIZE, IARG_END);
   }}
```

```
int main(int argc, char **argv) {
   PIN_Init(argc, argv);
   INS_AddInstrumentFunction(Instruction, 0);
   PIN_StartProgram();  // never returns
   return 0;
}
```

Instrumentation routine.

Memory read logger tool

```
#include “pin.h”
std::map<ADDRINT, std::string> disAssemblyMap;

void memoryRead(ADDRINT applicationIP,
   ADDRINT memoryReadAddress, UINT32 memoryReadSize) {
  printf(“0x%x %s reads %d bytes of memory at 0x%x\n”,
   applicationIP, disAssemblyMap[applicationIp].c_str(),
   memoryReadSize, memoryReadAddress);}

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)memoryRead,
           IARG_INST_PTR, // application IP
           IARG_MEMORYREAD_EA, // effective address of mem read
           IARG_MEMORY_READ_SIZE, IARG_END);
   }}

int main(int argc, char **argv) {
   PIN_Init(argc, argv);
   INS_AddInstrumentFunction(Instruction, 0);
   PIN_StartProgram();  // never returns
   return 0;
}
```

Memory read logger tool

```
void memoryRead(ADDRINT applicationIP,
    ADDRINT memoryReadAddress, UINT32 memoryReadSize) {
  printf(“0x%x %s reads %d bytes of memory at 0x%x\n”,
    applicationIP, disAssemblyMap[applicationIp].c_str(),
    memoryReadSize, memoryReadAddress);}

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)memoryRead,
              IARG_INST_PTR, // application IP
              IARG_MEMORYREAD_EA, // effective address of mem read
              IARG_MEMORY_READ_SIZE, IARG_END);
    }}
}
```

Memory read logger tool

```
void memoryRead(ADDRINT applicationIP,
    ADDRINT memoryReadAddress, UINT32 memoryReadSize) {
  printf("0x%x %s reads %d bytes of memory at 0x%x\n",
    applicationIP, disAssemblyMap[applicationIp].c_str(),
    memoryReadSize, memoryReadAddress);}

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)memoryRead,
              IARG_INST_PTR, // application IP
              IARG_MEMORYREAD_EA, // effective address of mem read
              IARG_MEMORY_READ_SIZE, IARG_END);
    }}
}
```

- **inc DWORD_PTR[%eax]**

Memory read logger tool

```
void memoryRead(ADDRINT applicationIP,
    ADDRINT memoryReadAddress, UINT32 memoryReadSize) {
  printf(“0x%x %s reads %d bytes of memory at 0x%x\n”,
    applicationIP, disAssemblyMap[applicationIp].c_str(),
    memoryReadSize, memoryReadAddress);}

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)memoryRead,
              IARG_INST_PTR, // application IP
              IARG_MEMORYREAD_EA, // effective address of mem read
              IARG_MEMORY_READ_SIZE, IARG_END);
    }}
}
```

```
Switch to pin stack
push 4
push %eax
push 0x7f083de
call memoryRead
Pop args off pin stack
Switch back to app stack
```

- inc DWORD_PTR[%eax]

Memory read logger tool

```
void memoryRead(ADDRINT applicationIP,
    ADDRINT memoryReadAddress, UINT32 memoryReadSize) {
  printf("0x%x %s reads %d bytes of memory at 0x%x\n",
    applicationIP, disAssemblyMap[applicationIp].c_str(),
    memoryReadSize, memoryReadAddress);}

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)memoryRead,
              IARG_INST_PTR, // application IP
              IARG_MEMORYREAD_EA, // effective address of mem read
              IARG_MEMORY_READ_SIZE, IARG_END);
    }}
}
```

```
Switch to pin stack
push 4
push %eax
push 0x7f083de
call memoryRead
Pop args off pin stack
Switch back to app stack
```

- inc DWORD_PTR[%eax]

- inc DWORD_PTR[%esi]0x8

Memory read logger tool

```
void memoryRead(ADDRINT applicationIP,
    ADDRINT memoryReadAddress, UINT32 memoryReadSize) {
  printf("0x%x %s reads %d bytes of memory at 0x%x\n",
    applicationIP, disAssemblyMap[applicationIp].c_str(),
    memoryReadSize, memoryReadAddress);}

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)memoryRead,
              IARG_INST_PTR, // application IP
              IARG_MEMORYREAD_EA, // effective address of mem reac
              IARG_MEMORY_READ_SIZE, IARG_END);
    }}
}
```

```
Switch to pin stack
push 4
push %eax
push 0x7f083de
call memoryRead
Pop args off pin stack
Switch back to app stack
```

- inc DWORD_PTR[%eax]

```
Switch to pin stack
push 4
lea  %ecx,[%esi]0x8
push %ecx
push 0x7f083e4
call memoryRead
Pop args off pin stack
Switch back to app stack
```

- inc DWORD_PTR[%esi]0x8

Memory read logger tool

```
void memoryRead(ADDRINT applicationIP,
    ADDRINT memoryReadAddress, UINT32 memoryReadSize) {
  printf("0x%x %s reads %d bytes of memory at 0x%x\n",
    applicationIP, disAssemblyMap[applicationIp].c_str(),
    memoryReadSize, memoryReadAddress);}

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)memoryRead,
              IARG_INST_PTR, // application IP
              IARG_MEMORYREAD_EA, // effective address of mem reac
              IARG_MEMORY_READ_SIZE, IARG_END);
    }}
}
```

```
Switch to pin stack
push 4
push %eax
push 0x7f083de
call memoryRead
Pop args off pin stack
Switch back to app stack
```

- inc DWORD_PTR[%eax]

```
Switch to pin stack
push 4
lea  %ecx,[%esi]0x8
push %ecx
push 0x7f083e4
call memoryRead
Pop args off pin stack
Switch back to app stack
```

- inc DWORD_PTR[%esi]0x8

Pin has determined that it can overwrite ecx

Memory read logger tool

```
void memoryRead(ADDRINT applicationIP,
    ADDRINT memoryReadAddress, UINT32 memoryReadSize) {
  printf(“0x%x %s reads %d bytes of memory at 0x%x\n”,
    applicationIP, disAssemblyMap[applicationIp].c_str(),
    memoryReadSize, memoryReadAddress);}

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)memoryRead,
              IARG_INST_PTR, // application IP
              IARG_MEMORYREAD_EA, // effective address of mem reac
              IARG_MEMORY_READ_SIZE, IARG_END);
    }}
}
```

```
Switch to pin stack
push 4
push %eax
push 0x7f083de
call memoryRead
Pop args off pin stack
Switch back to app stack
```

- inc DWORD_PTR[%eax]

```
Switch to pin stack
push 4
lea  %ecx,[%esi]0x8
push %ecx
push 0x7f083e4
call memoryRead
Pop args off pin stack
Switch back to app stack
```

- inc DWORD_PTR[%esi]0x8

Memory read logger tool

```
#include “pin.h”
std::map<ADDRINT, std::string> disAssemblyMap;

void memoryRead(ADDRINT applicationIP,
   ADDRINT memoryReadAddress, UINT32 memoryReadSize) {
  printf(“0x%x %s reads %d bytes of memory at 0x%x\n”,
   applicationIP, disAssemblyMap[applicationIp].c_str(),
   memoryReadSize, memoryReadAddress);}

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)memoryRead,
            IARG_INST_PTR, // application IP
            IARG_MEMORYREAD_EA, // effective address of mem read
            IARG_MEMORY_READ_SIZE, IARG_END);
   }}

int main(int argc, char **argv) {
   PIN_Init(argc, argv);
   INS_AddInstrumentFunction(Instruction, 0);
   PIN_StartProgram();  // never returns
   return 0;
}
```

Memory read logger tool

```
#include “pin.h”
std::map<ADDRINT, std::string> disAssemblyMap;

void memoryRead(ADDRINT applicationIP,
   ADDRINT memoryReadAddress, UINT32 memoryReadSize) {
  printf(“0x%x %s reads %d bytes of memory at 0x%x\n”,
   applicationIP, disAssemblyMap[applicationIp].c_str(),
   memoryReadSize, memoryReadAddress);}

void Instruction(INS ins, void *v) { // jitting time routine
   if (INS_IsMemoryRead(ins)) {
       disAssemblyMap[INS_Address(ins)]
       INS_InsertCall(ins, IPOI
           IARG_INST_PTR, // applicati
           IARG_MEMORYREAD_EA, // effec
           IARG_MEMORY_READ_SIZE, IARG_
   }}

int main(int argc, char **argv) {
   PIN_Init(argc, argv);
   INS_AddInstrumentFunction(Instruction, 0);
   PIN_StartProgram();  // never returns
   return 0;
}
```

INS_IsMemoryRead

True if the instruction reads memory.

Memory read logger tool

```
#include “pin.h”
std::map<ADDRINT, std::string> disAssemblyMap;

void memoryRead(ADDRINT applicationIP,
   ADDRINT memoryReadAddress, UINT32 memoryReadSize) {
  printf(“0x%x %s reads %d bytes of memory at 0x%x\n”,
   applicationIP, disAssemblyMap[applicationIp].c_str(),
   memoryReadSize, memoryReadAddress);}

void Instruction(INS ins, void *v) { // jitting time routine
   if (INS_IsMemoryRead(ins)) {
       disAssemblyMap[              (ins)]
       INS_InsertCall(ins, IPOI
           IARG_INST_PTR, // applicati
           IARG_MEMORYREAD_EA, // effec
           IARG_MEMORY_READ_SIZE, IARG_
   }}

int main(int argc, char **argv) {
   PIN_Init(argc, argv);
   INS_AddInstrumentFunction(Instruction, 0);
   PIN_StartProgram();  // never returns
   return 0;
}
```

INS_IsMemoryWrite

True if the instruction writes memory.

Memory read logger tool

```
#include “pin.h”
std::map<ADDRINT, std::string> disAssemblyMap;

void memoryRead(ADDRINT applicationIP,
   ADDRINT memoryReadAddress, UINT32 memoryReadSize) {
  printf(“0x%x %s reads %d bytes of memory at 0x%x\n”,
   applicationIP, disAssemblyMap[applicationIp].c_str(),
   memoryReadSize, memoryReadAddress);}

void Instruction(INS ins, void *v) { // jitting time routine
   if (INS_IsMemoryRead(ins)) {
       disAssemblyMap[    ddress(ins)]
       INS_InsertCall(ins, IPOI
           IARG_INST_PTR, // applicati
           IARG_MEMORYREAD_EA, // effec
           IARG_MEMORY_READ_SIZE, IARG_
   }}

int main(int argc, char **argv) {
   PIN_Init(argc, argv);
   INS_AddInstrumentFunction(Instruction, 0);
   PIN_StartProgram();  // never returns
   return 0;
}
```

INS_IsNop

True if the instruction is a nop.

Memory read logger tool

```
#include “pin.h”
std::map<ADDRINT, std::string> disAssemblyMap;

void memoryRead(ADDRINT applicationIP,
   ADDRINT memoryReadAddress, UINT32 memoryReadSize) {
  printf(“0x%x %s reads %d bytes of memory at 0x%x\n”,
   applicationIP, disAssemblyMap[applicationIp].c_str(),
   memoryReadSize, memoryReadAddress);}

void Instruction(INS ins, void *v) { // jitting time routine
   if (INS_IsMemoryRead(ins)) {
       disAssemblyMap[               (ins)]
       INS_InsertCall(ins, IPO
           IARG_INST_PTR, // applicati
           IARG_MEMORYREAD_EA, // effec
           IARG_MEMORY_READ_SIZE, IARG_
   }}

int main(int argc, char **argv) {
   PIN_Init(argc, argv);
   INS_AddInstrumentFunction(Instruction, 0);
   PIN_StartProgram();  // never returns
   return 0;
}
```

INS_IsProcedureCall

True if the instruction is a procedure call.

Memory read logger tool

```
#include “pin.h”
std::map<ADDRINT, std::string> disAssemblyMap;

void memoryRead(ADDRINT applicationIP,
   ADDRINT memoryReadAddress, UINT32 memoryReadSize) {
  printf(“0x%x %s reads %d bytes of memory at 0x%x\n”,
   applicationIP, disAssemblyMap[applicationIp].c_str(),
   memoryReadSize, memoryReadAddress);}

void Instruction(INS ins, void *v) { // jitting time routine
   if (INS_IsMemoryRead(ins)) {
       disAssemblyMap[          ddress(ins)]
       INS_InsertCall(ins, IPO
           IARG_INST_PTR, // applicati
           IARG_MEMORYREAD_EA, // effec
           IARG_MEMORY_READ_SIZE, IARG_
   }}

int main(int argc, char **argv) {
   PIN_Init(argc, argv);
   INS_AddInstrumentFunction(Instruction, 0);
   PIN_StartProgram();  // never returns
   return 0;
}
```

INS_IsRet

True if the instruction is a return instruction.

Memory read logger tool

```
#include “pin.h”
std::map<ADDRINT, std::string> disAssemblyMap;

void memoryRead(ADDRINT applicationIP,
   ADDRINT memoryReadAddress, UINT32 memoryReadSize) {
  printf(“0x%x %s reads %d bytes of memory at 0x%x\n”,
   applicationIP, disAssemblyMap[applicationIp].c_str(),
   memoryReadSize, memoryReadAddress);}

void Instruction(INS ins, void *v) { // jitting time routine
   if (INS_IsMemoryRead(ins)) {
       disAssemblyMap[           ddress(ins)]
       INS_InsertCall(ins, IPOI
           IARG_INST_PTR, // applicati
           IARG_MEMORYREAD_EA, // effec
           IARG_MEMORY_READ_SIZE, IARG_
   }}

int main(int argc, char **argv) {
   PIN_Init(argc, argv);
   INS_AddInstrumentFunction(Instruction, 0);
   PIN_StartProgram();  // never returns
   return 0;
}
```

INS_IsSyscall

True if the instruction is a system call.

Memory read logger tool

```
#include “pin.h”
std::map<ADDRINT, std::string> disAssemblyMap;

void memoryRead(ADDRINT applicationIP,
   ADDRINT memoryReadAddress, UINT32 memoryReadSize) {
  printf(“0x%x %s reads %d bytes of memory at 0x%x\n”,
   applicationIP, disAssemblyMap[applicationIp].c_str(),
   memoryReadSize, memoryReadAddress);}

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)memoryRead,
           IARG_INST_PTR, // application IP
           IARG_MEMORYREAD_EA, // effective address of mem read
           IARG_MEMORY_READ_SIZE, IARG_END);
   }}

int main(int argc, char **argv) {
   PIN_Init(argc, argv);
   INS_AddInstrumentFunction(Instruction, 0);
   PIN_StartProgram();  // never returns
   return 0;
}
```

Memory read logger tool

```
#include “pin.h”
std::map<ADDRINT, std::string> disAssemblyMap;

void memoryRead(ADDRINT applicationIP,
   ADDRINT memoryReadAddress, UINT32 memoryReadSize) {
  printf(“0x%x %s reads %d bytes of memory at 0x%x\n”,
   applicationIP, disAssemblyMap[applicationIp].c_str(),
   memoryReadSize, memoryReadAddress);}

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)memoryRead,
           IARG_INST_PTR, // application IP
           IARG_MEMORYREAD_EA, // effective address of mem read
           IARG_MEMORY_READ_SIZE, IARG_END);
   }}

int main(int argc, char **argv) {
   PIN_Init(argc, argv);
   INS_AddInstrumentFunction(Instruction, 0);
   PIN_StartProgram();  // never returns
   return 0;
}
```

Memory read logger tool

```
#include “pin.h”
std::map<ADDRINT, std::string> disAssemblyMap;

void memoryRead(ADDRINT applicationIP,
   ADDRINT memoryReadAddress, UINT32 memoryReadSize) {
  printf(“0x%x %s reads %d bytes of memory at 0x%x\n”,
   applicationIP, disAssemblyMap[applicationIp].c_str(),
   memoryReadSize, memoryReadAddress);}

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)memoryRead,
           IARG_INST_PTR, // application IP
           IARG_MEMORYREAD_EA, // effective address of mem read
           IARG_MEMORY_READ_SIZE, IARG_END);
   }}

int main(int argc, char **argv) {
   PIN_Init(argc, argv);
   INS_AddInstrumentFunction(Instruction, 0);
   PIN_StartProgram();  // never returns
   return 0;
}
```

Many more IARG_ possible.

Memory read logger tool

```
#include “pin.h”
std::map<ADDRINT, std::string> disAssemblyMap;

void memoryRead(ADDRINT applicationIP,
   ADDRINT memoryReadAddress, UINT32 memoryReadSize) {
  printf(“0x%x %s reads %d bytes of memory at 0x%x\n”,
   applicationIP, disAssemblyMap[applicationIp].c_str(),
   memoryReadSize, memoryReadAddress);}

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)memoryRead,
           IARG_INST_PTR, // application IP
           IARG_MEMORYREAD_EA, // effective address of mem read
           IARG_MEMORY_READ_SIZE, IARG_END);
   }}

int main(int argc, char **argv) {
   PIN_Init(argc, argv);
   INS_AddInstrumentFunction(Instructio
   PIN_StartProgram();  // never return
   return 0;
}
```

IARG_REG_VALUE, <REG>

Value of a register.

Memory read logger tool

```
#include “pin.h”
std::map<ADDRINT, std::string> disAssemblyMap;

void memoryRead(ADDRINT applicationIP,
    ADDRINT memoryReadAddress, UINT32 memoryReadSize) {
  printf(“0x%x %s reads %d bytes of memory at 0x%x\n”,
    applicationIP, disAssemblyMap[applicationIp].c_str(),
    memoryReadSize, memoryReadAddress);}

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)memoryRead,
            IARG_INST_PTR, // application IP
            IARG_MEMORYREAD_EA, // effective address of mem read
            IARG_MEMORY_READ_SIZE, IARG_END);
    }}

int main(int argc, char **argv) {
    PIN_Init(argc, argv);
    INS_AddInstrumentFunction(Instructio
    PIN_StartProgram();  // never return
    return 0;
}
```

IARG_BRANCH_TARGET_ADDR

Target address of this branch instruction, only valid if INS_IsBranchOrCall is true

Memory read logger tool

```
#include “pin.h”
std::map<ADDRINT, std::string> disAssemblyMap;

void memoryRead(ADDRINT applicationIP,
   ADDRINT memoryReadAddress, UINT32 memoryReadSize) {
  printf(“0x%x %s reads %d bytes of memory at 0x%x\n”,
   applicationIP, disAssemblyMap[applicationIp].c_str(),
   memoryReadSize, memoryReadAddress);}

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)memoryRead,
           IARG_INST_PTR, // application IP
           IARG_MEMORYREAD_EA, // effective address of mem read
           IARG_MEMORY_READ_SIZE, IARG_END);
   }}

int main(int argc, char **argv) {
   PIN_Init(argc, argv);
   INS_AddInstrumentFunction(Instruction
   PIN_StartProgram();  // never returns
   return 0;
}
```

IARG_FUNCARG_ENTRYPOINT_VALUE, <ARG #>

The value of #th arg of the function. (Valid at the callsite.)

Memory read logger tool

```
#include “pin.h”
std::map<ADDRINT, std::string> disAssemblyMap;

void memoryRead(ADDRINT applicationIP,
   ADDRINT memoryReadAddress, UINT32 memoryReadSize) {
  printf(“0x%x %s reads %d bytes of memory at 0x%x\n”,
   applicationIP, disAssemblyMap[applicationIp].c_str(),
   memoryReadSize, memoryReadAddress);}

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)memoryRead,
           IARG_INST_PTR, // application IP
           IARG_MEMORYREAD_EA, // effective address of mem read
           IARG_MEMORY_READ_SIZE, IARG_END);
   }}

int main(int argc, char **argv) {
   PIN_Init(argc, argv);
   INS_AddInstrumentFunction(Instructio
   PIN_StartProgram();  // never returns
   return 0;
}
```

Work in progress:

IARG_MAKE_ME_A_COFFEE

Example 4: Malloc wrapping

Malloc tracing tool

```
#include “pin.h”

void mallocBefore(ADDRINT size){ printf(“malloc(%d)\n”, size); }
void mallocAfter(ADDRINT ret){ printf(“\tmalloc returns 0x%x\n”, ret); }

void Image(IMG img, void *v) { // jitting time routine
   RTN mallocRtn = RTN_FindByName(img, “malloc”);
   if (RTN_Valid(mallocRtn) {
            RTN_Open(mallocRtn);
            RTN_InsertCall(mallocRtn, IPOINT_BEFORE, (AFUNPTR)mallocBefore,
                IARG_FUNCARG_ENTRYPOINT_VALUE, 0, IARG_END);
            RTN_InsertCall(mallocRtn, IPOINT_AFTER, (AFUNPTR)mallocAfter,
                IARG_FUNCARG_EXITPOINT_VALUE, IARG_END);
            RTN_Close(mallocRtn);
   } }

int main(int argc, char **argv) {
   PIN_InitSymbols();
   PIN_Init(argc, argv);
   IMG_AddInstrumentFunction(Image, 0);
   PIN_StartProgram();  // never returns
   return 0;
}
```

Malloc tracing tool

```
#include “pin.h”

void mallocBefore(ADDRINT size){ printf(“malloc(%d)\n”, size); }
void mallocAfter(ADDRINT ret){ printf(“\tmalloc returns 0x%x\n”, ret); }

void Image(IMG img, void *v) { // jitting time routine
   RTN mallocRtn = RTN_FindByName(img, “malloc”);
   if (RTN_Valid(mallocRtn) {
           RTN_Open(mallocRtn);
           RTN_InsertCall(mallocRtn, IPOINT_BEFORE, (AFUNPTR)mallocBefore,
               IARG_FUNCARG_ENTRYPOINT_VALUE, 0, IARG_END);
           RTN_InsertCall(mallocRtn, IPOINT_AFTER, (AFUNPTR)mallocAfter,
               IARG_FUNCARG_EXITPOINT_VALUE, IARG_END);
           RTN_Close(mallocRtn);
   } }

int main(int argc, char **argv) {
   PIN_InitSymbols();
   PIN_Init(argc, argv);
   IMG_AddInstrumentFunction(Image, 0);
   PIN_StartProgram();  // never returns
   return 0;
}
```

Malloc tracing tool

```
#include “pin.h”

void mallocBefore(ADDRINT size){ printf(“malloc(%d)\n”, size); }
void mallocAfter(ADDRINT ret){ printf(“\tmalloc returns 0x%x\n”, ret); }

void Image(IMG img, void *v) { // jitting time routine
   RTN mallocRtn = RTN_FindByName(img, “malloc”);
   if (RTN_Valid(mallocRtn) {
           RTN_Open(mallocRtn);
           RTN_InsertCall(mallocRtn, IPOINT_BEFORE, (AFUNPTR)mallocBefore,
               IARG_FUNCARG_ENTRYPOINT_VALUE, 0, IARG_END);
           RTN_InsertCall(mallocRtn, IPOINT_AFTER, (AFUNPTR)mallocAfter,
               IARG_FUNCARG_EXITPOINT_VALUE, IARG_END);
           RTN_Close(mallocRtn);
   } }

int main(int argc, char **argv) {
   PIN_InitSymbols();
   PIN_Init(argc, argv);
   IMG_AddInstrumentFunction(Image, 0);
   PIN_StartProgram();  // never returns
   return 0;
}
```

Malloc tracing tool

```
#include “pin.h”

void mallocBefore(ADDRINT size){ printf(“malloc(%d)\n”, size); }
void mallocAfter(ADDRINT ret){ printf(“\tmalloc returns 0x%x\n”, ret); }

void Image(IMG img, void *v) { // jitting time routine
   RTN mallocRtn = RTN_FindByName(img, “malloc”);
   if (RTN_Valid(mallocRtn) {
           RTN_Open(mallocRtn);
           RTN_InsertCall(mallocRtn, IPOINT_BEFORE, (AFUNPTR)mallocBefore,
               IARG_FUNCARG_ENTRYPOINT_VALUE, 0, IARG_END);
           RTN_InsertCall(mallocRtn, IPOINT_AFTER, (AFUNPTR)mallocAfter,
               IARG_FUNCARG_EXITPOINT_VALUE, IARG_END);
           RTN_Close(mallocRtn);
   } }

int main(int argc, char **argv) {
   PIN_InitSymbols();
   PIN_Init(argc, argv);
   IMG_AddInstrumentFunction(Image, 0);
   PIN_StartProgram();  // never returns
   return 0;
}
```

Instruct Pin to make use of any symbols which are available for this process.

Malloc tracing tool

```
#include “pin.h”

void mallocBefore(ADDRINT size){ printf(“malloc(%d)\n”, size); }
void mallocAfter(ADDRINT ret){ printf(“\tmalloc returns 0x%x\n”, ret); }

void Image(IMG img, void *v) { // jitting time routine
   RTN mallocRtn = RTN_FindByName(img, “malloc”);
   if (RTN_Valid(mallocRtn) {
           RTN_Open(mallocRtn);
           RTN_InsertCall(mallocRtn, IPOINT_BEFORE, (AFUNPTR)mallocBefore,
               IARG_FUNCARG_ENTRYPOINT_VALUE, 0, IARG_END);
           RTN_InsertCall(mallocRtn, IPOINT_AFTER, (AFUNPTR)mallocAfter,
               IARG_FUNCARG_EXITPOINT_VALUE, IARG_END);
           RTN_Close(mallocRtn);
   } }

int main(int argc, char **argv) {
   PIN_InitSymbols();
   PIN_Init(argc, argv);
   IMG_AddInstrumentFunction(Image, 0);
   PIN_StartProgram();  // never returns
   return 0;
}
```

Direct Pin to call `Image` whenever an image is loaded, it can be a library or the main exec file.

Malloc tracing tool

```
#include “pin.h”

void mallocBefore(ADDRINT size){ printf(“malloc(%d)\n”, size); }
void mallocAfter(ADDRINT ret){ printf(“\tmalloc returns 0x%x\n”, ret); }

void Image(IMG img, void *v) { // jitting time routine
   RTN mallocRtn = RTN_FindByName(img, “malloc”);
   if (RTN_Valid(mallocRtn) {
           RTN_Open(mallocRtn);
           RTN_InsertCall(mallocRtn, IPOINT_BEFORE, (AFUNPTR)mallocBefore,
               IARG_FUNCARG_ENTRYPOINT_VALUE, 0, IARG_END);
           RTN_InsertCall(mallocRtn, IPOINT_AFTER, (AFUNPTR)mallocAfter,
               IARG_FUNCARG_EXITPOINT_VALUE, IARG_END);
           RTN_Close(mallocRtn);
   } }

int main(int argc, char **argv) {
   PIN_InitSymbols();
   PIN_Init(argc, argv);
   IMG_AddInstrumentFunction(Image, 0);
   PIN_StartProgram();  // never returns
   return 0;
}
```

Instrumentation routine.

Malloc tracing t

A handle to the image being loaded.

```
#include “pin.h”

void mallocBefore(ADDRINT size){ printf(“malloc(%d)\n”, size); }
void mallocAfter(ADDRINT ret){ printf(“\tmalloc returns 0x%x\n”, ret); }

void Image(IMG img, void *v) { // jitting time routine
   RTN mallocRtn = RTN_FindByName(img, “malloc”);
   if (RTN_Valid(mallocRtn) {
           RTN_Open(mallocRtn);
           RTN_InsertCall(mallocRtn, IPOINT_BEFORE, (AFUNPTR)mallocBefore,
               IARG_FUNCARG_ENTRYPOINT_VALUE, 0, IARG_END);
           RTN_InsertCall(mallocRtn, IPOINT_AFTER, (AFUNPTR)mallocAfter,
               IARG_FUNCARG_EXITPOINT_VALUE, IARG_END);
           RTN_Close(mallocRtn);
   } }

int main(int argc, char **argv) {
   PIN_InitSymbols();
   PIN_Init(argc, argv);
   IMG_AddInstrumentFunction(Image, 0);
   PIN_StartProgram();  // never returns
   return 0;
}
```

Instrumentation routine.

Malloc tracing t

A handle to the image being loaded.

```
#include “pin.h”

void mallocBefore(ADDRINT size){ printf(“malloc(%d)\n”, size); }
void mallocAfter(ADDRINT ret){ printf(“\tmalloc returns 0x%x\n”, ret); }

void Image(IMG img, void *v) { // jitting time routine
   RTN mallocRtn = RTN_FindByName(img, “malloc”);
   if (RTN_Valid(mallocRtn) {
           RTN_Open(mallocRtn);
           RTN_InsertCall(mallocRtn, IPOINT_BEFORE, (AFUNPTR)mallocBefore,
               IARG_FUNCARG_ENTRYPOINT_VALUE, 0, IARG_END);
           RTN_InsertCall(mallocRtn, IPOINT_AFTER, (AFUNPTR)mallocAfter,
               IARG_FUNCARG_EXITPOINT_VALUE, IARG_END);
           RTN_Close(mallocRtn);
   } }

int main(int argc, char **argv) {
   PIN_InitSymbols();
   PIN_Init(argc, argv);
   IMG_AddInstrumentFunction(Image, 0);
   PIN_StartProgram();  // never returns
   return 0;
}
```

Instrumentation routine.

Malloc tracing tool

```
#include “pin.h”

void mallocBefore(ADDRINT size){ printf(“malloc(%d)\n”, size); }
void mallocAfter(ADDRINT ret){ printf(“\tmalloc returns 0x%x\n”, ret); }

void Image(IMG img, void *v) { // jitting time routine
   RTN mallocRtn = RTN_FindByName(img, “malloc”);
   if (RTN_Valid(mallocRtn) {
            RTN_Open(mallocRtn);
            RTN_InsertCall(mallocRtn, IPOINT_BEFORE, (AFUNPTR)mallocBefore,
                IARG_FUNCARG_ENTRYPOINT_VALUE, 0, IARG_END);
            RTN_InsertCall(mallocRtn, IPOINT_AFTER, (AFUNPTR)mallocAfter,
                IARG_FUNCARG_EXITPOINT_VALUE, IARG_END);
            RTN_Close(mallocRtn);
   } }

int main(int argc, char **argv) {
   PIN_InitSymbols();
   PIN_Init(argc, argv);
   IMG_AddInstrumentFunction(Image, 0);
   PIN_StartProgram();  // never returns
   return 0;
}
```

Instrumentation routine.

Malloc tracing t

```
#include “pin.h”

void mallocBefore(ADDRINT size){ printf(“malloc(%d)\n”, size); }
void mallocAfter(ADDRINT ret){ printf(“\tmalloc returns 0x%x\n”, ret); }

void Image(IMG img, void *v) { // jitting time routine
   RTN mallocRtn = RTN_FindByName(img, “malloc”);
   if (RTN_Valid(mallocRtn) {
           RTN_Open(mallocRtn);
           RTN_InsertCall(mallocRtn, IPOINT_BEFORE, (AFUNPTR)mallocBefore,
               IARG_FUNCARG_ENTRYPOINT_VALUE, 0, IARG_END);
           RTN_InsertCall(mallocRtn, IPOINT_AFTER, (AFUNPTR)mallocAfter,
               IARG_FUNCARG_EXITPOINT_VALUE, IARG_END);
           RTN_Close(mallocRtn);
   } }

int main(int argc, char **argv) {
   PIN_InitSymbols();
   PIN_Init(argc, argv);
   IMG_AddInstrumentFunction(Image, 0);
   PIN_StartProgram();  // never returns
   return 0;
}
```

Analysis routines.

Instrumentation routine.

Malloc tracing t

Analysis routines.

```
#include “pin.h”

void mallocBefore(ADDRINT size){ printf(“malloc(%d)\n”, size); }
void mallocAfter(ADDRINT ret){ printf(“\tmalloc returns 0x%x\n”, ret); }

void Image(IMG img, void *v) { // jitting time routine
   RTN mallocRtn = RTN_FindByName(img, “malloc”);
   if (RTN_Valid(mallocRtn) {
           RTN_Open(mallocRtn);
           RTN_InsertCall(mallocRtn, IPOINT_BEFORE, (AFUNPTR)mallocBefore,
               IARG_FUNCARG_ENTRYPOINT_VALUE, 0, IARG_END);
           RTN_InsertCall(mallocRtn, IPOINT_AFTER, (AFUNPTR)mallocAfter,
               IARG_FUNCARG_EXITPOINT_VALUE, IARG_END);
           RTN_Close(mallocRtn);
   } }

int main(int argc, char **argv) {
   PIN_InitSymbols();
   PIN_Init(argc, argv);
   IMG_AddInstrumentFunction(Image, 0);
   PIN_StartProgram();  // never returns
   return 0;
}
```

Instrumentation routine.