Dynamic Binary Instrumentation

Zhiqiang Lin

Department of Computer Science and Engineering
The Ohio State University

7/19/2018

Outline

Outline

What Is Instrumentation

A technique that inserts extra code into a program to collect information of your interest. Such technique has been widely used in practice in both program debugging and security analysis.

What Is Instrumentation

A technique that inserts extra code into a program to collect information of your interest. Such technique has been widely used in practice in both program debugging and security analysis.

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

What Is Instrumentation

A technique that inserts extra code into a program to collect information of your interest. Such technique has been widely used in practice in both program debugging and security analysis.

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

What Is (Dynamic) Binary Instrumentation

A technique that inserts extra code into the binary code of a program to collect (runtime) information of your interest.

What Is (Dynamic) Binary Instrumentation

A technique that inserts extra code into the binary code of a program to collect (runtime) information of your interest.

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

What Can Instrumentation Do?

- Profiler for compiler optimization:
  - Basic-block count
  - Value profile
- Micro architectural study:
  - Instrument branches to simulate branch predictors
  - Generate traces
- Bug checking/Vulnerability identification/Exploit generation:
  - Find references to uninitialized, unallocated address
  - Inspect argument at particular function call
  - Inspect function pointers and return addresses
- Software tools that use dynamic binary instrumentation:
  - Valgrind, Pin, QEMU, DynInst, ...

Instrumentation approaches: source vs. binary

- Source instrumentation:
  - Instrument source programs
- Binary instrumentation:
  - Instrument executables directly
- Advantages for binary instrumentation
  - Language independent
  - Machine-level view
  - Instrument legacy/proprietary software

Instrumentation approaches: static vs. dynamic

- When to instrument
  - Instrument statically - before runtime
  - Instrument dynamically - during runtime
- Advantages for dynamic instrumentation
  - No need to recompile or relink
  - Discover code at runtime
  - Handle dynamically-generated code
  - Attach to running processes

How is Instrumentation used in Program Analysis?

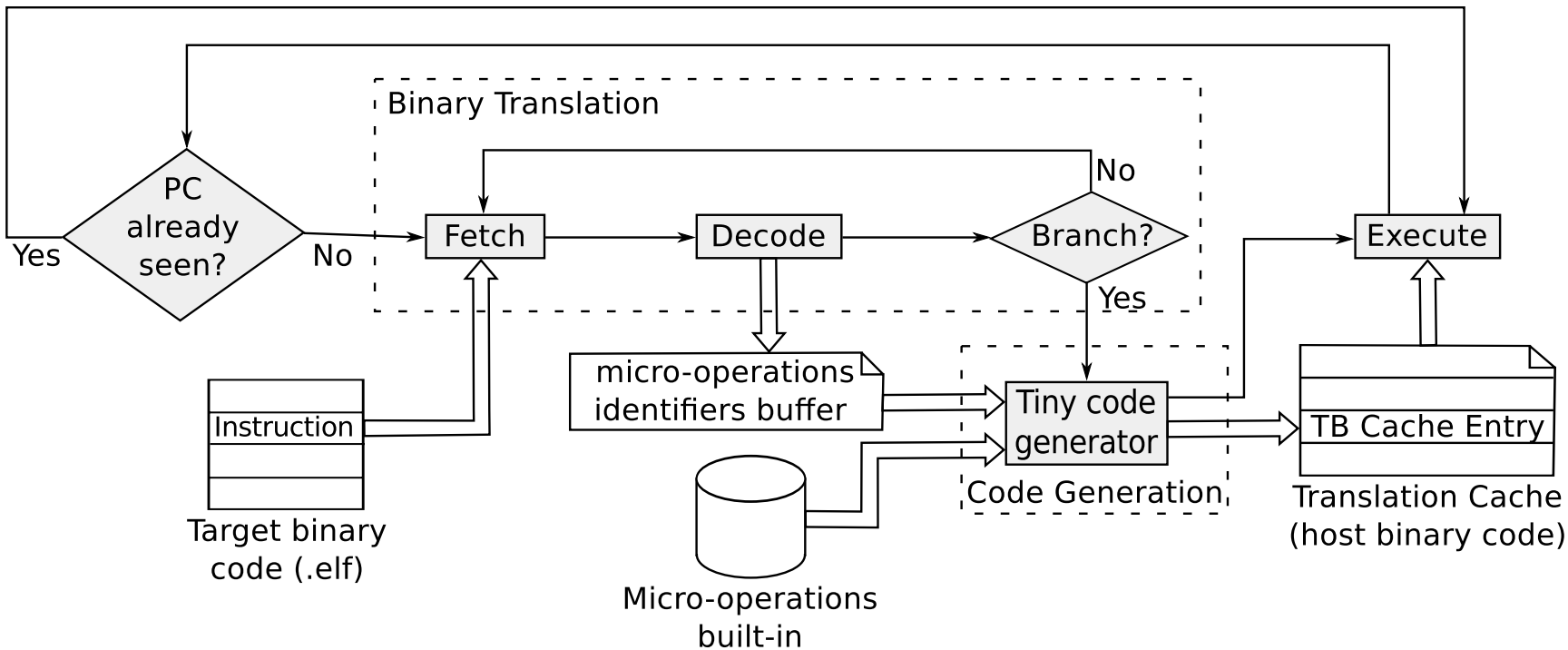
- Code coverage
- Call-graph generation
- Memory-leak detection
- Vulnerability identification
- Instruction profiling
- Data dependence profiling
- Thread analysis
  - Thread profiling
  - Race detection

Outline

QEMU

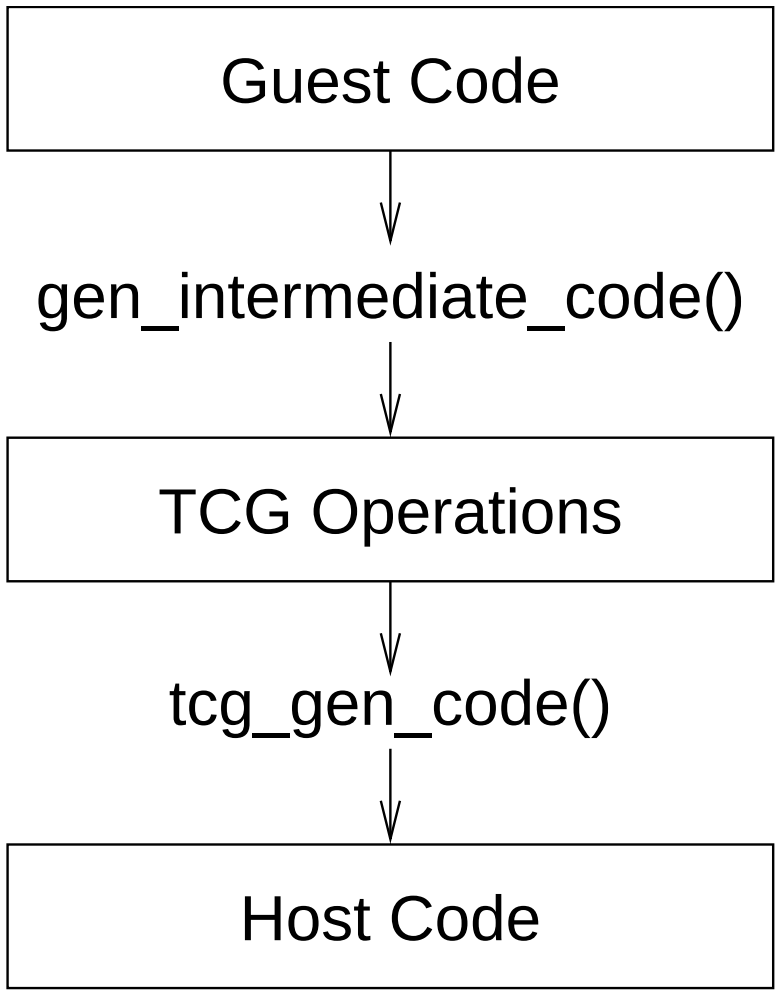
- QEMU is a generic and open source machine emulator and virtualizer.
- As a machine emulator, QEMU can run OSes and programs made for one machine (e.g. an ARM board) on a different machine (e.g. your own PC). By using dynamic translation, it achieves very good performance.
- As a virtualizer, QEMU achieves near native performances by executing the guest code directly on the host CPU. QEMU supports virtualization when executing under the Xen hypervisor or using the KVM kernel module in Linux. When using KVM, QEMU can virtualize x86, server and embedded PowerPC, and S390 guests.

QEMU Internals

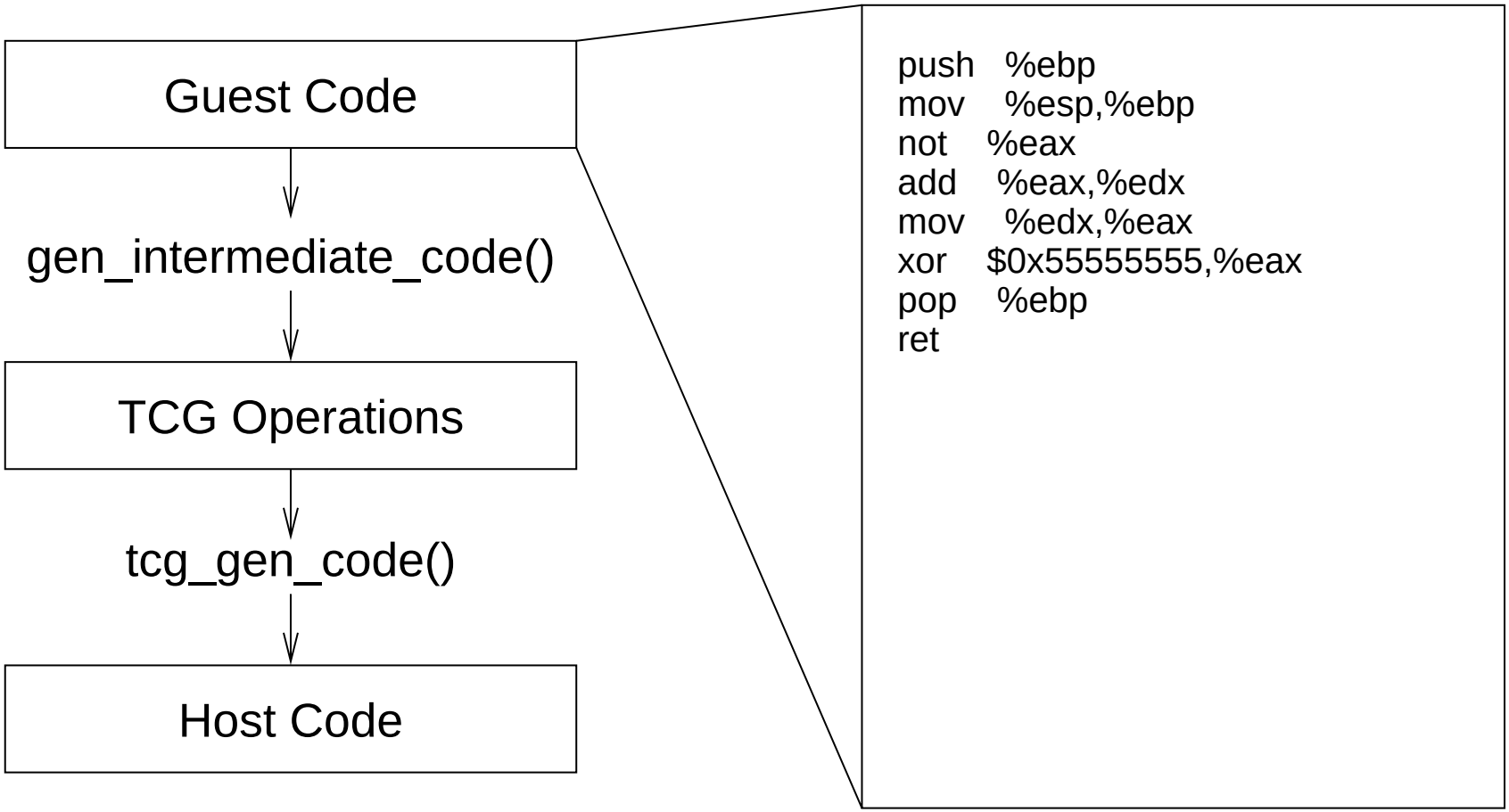


QEMU-Code Translation

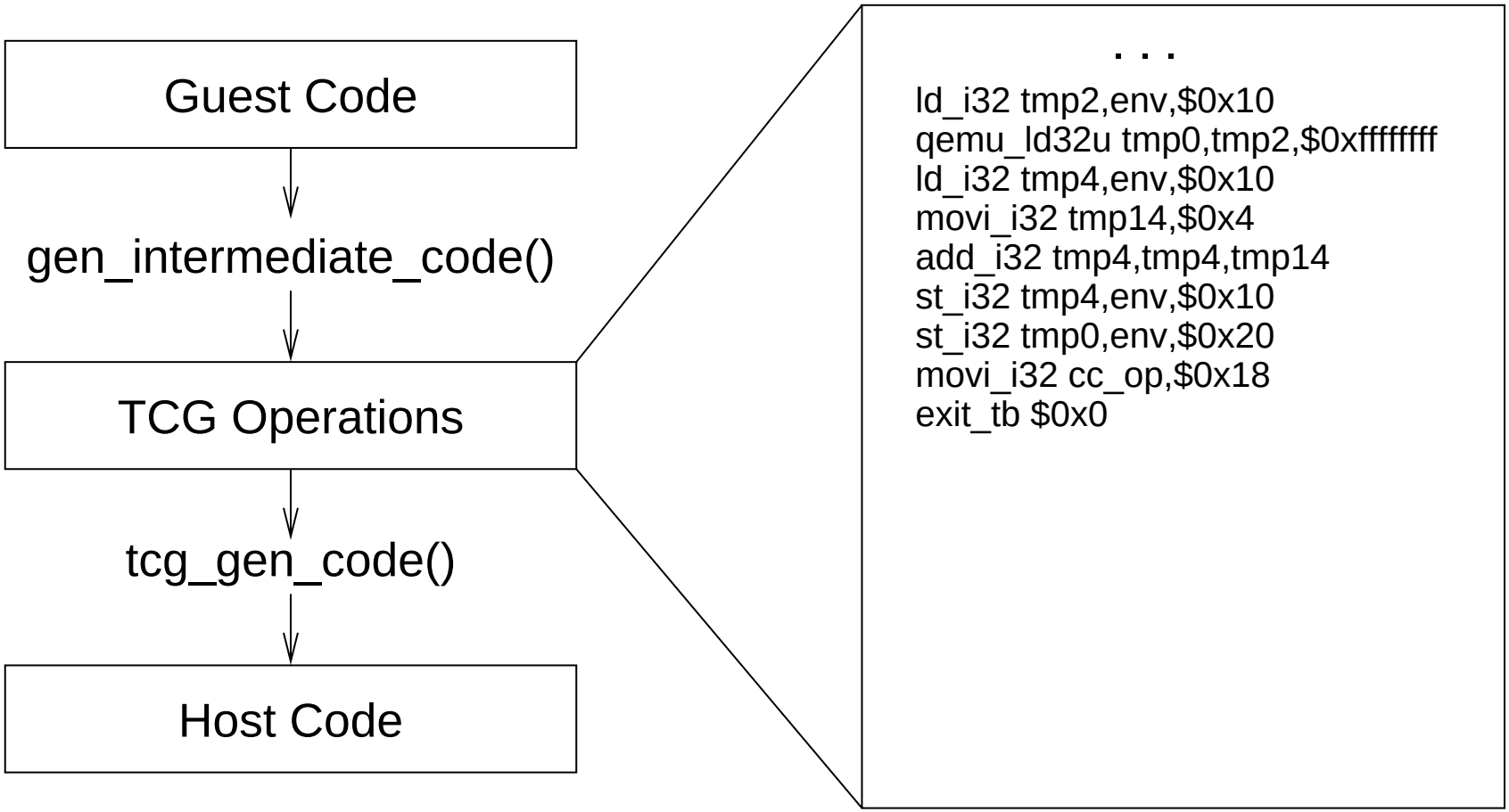
- QEMU uses an intermediate form.
- Frontends are in target-*/, includes alpha, arm, cris, i386, m68k,mips, ppc, sparc, etc.
- Backends are in tcg/*, includes arm/, hppa/, i386/, ia64/, mips/, ppc/, ppc64/, s390/, sparc/, tcg.c, tcg.h, tcg-opc.h, tcg-op.h, tcg-runtime.h



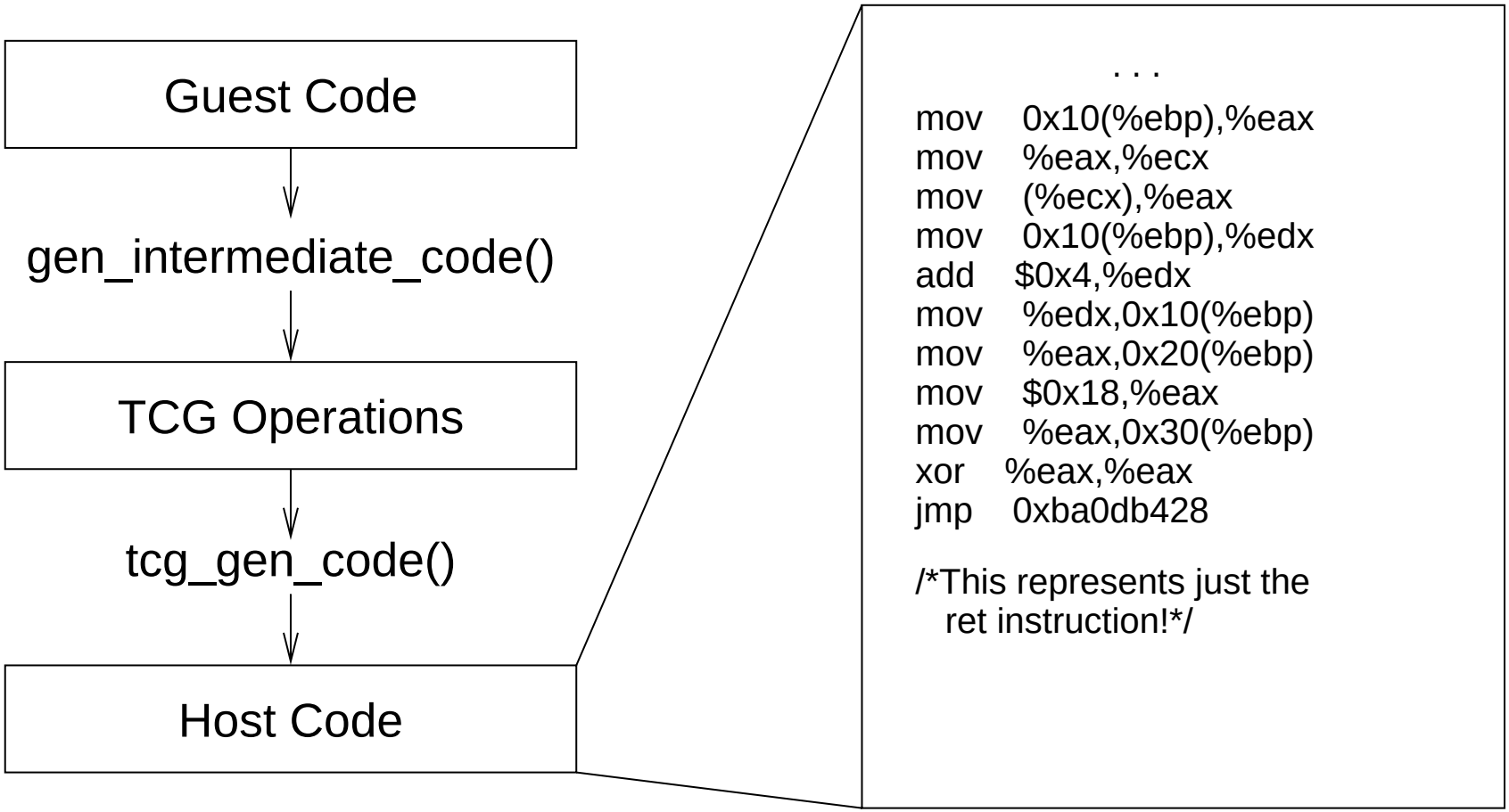
QEMU-Code Translation



QEMU-Code Translation



QEMU-Code Translation



QEMU-Code Base

- TranslationBlock structure in `translate-all.h`
- Translation cache is code gen buffer in `exec.c`
- `cpu-exec()` in cpu-exec.c orchestrates translation and block chaining.
- `target-*/translate.c`: guest ISA specific code.
- `tcg-*/*/`: host ISA specific code.
- `linux-user/*`: Linux usermode specific code.
- `vl.c`: Main loop for system emulation.
- `hw/*`: Hardware, including video, audio, and boards.

QEMU use cases

- Malware analysis
- Dynamic binary code instrumentation
- System wide taint analysis
- System wide data lifetime tracking
- Being part of KVM
- Execution replay
- ...

Outline

PIN

- Pin is a tool for the instrumentation of programs. It supports Linux* and Windows* executables for IA-32, Intel(R) 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.

Credit: The rest of the slides are compiled from Intel's PIN tutorial

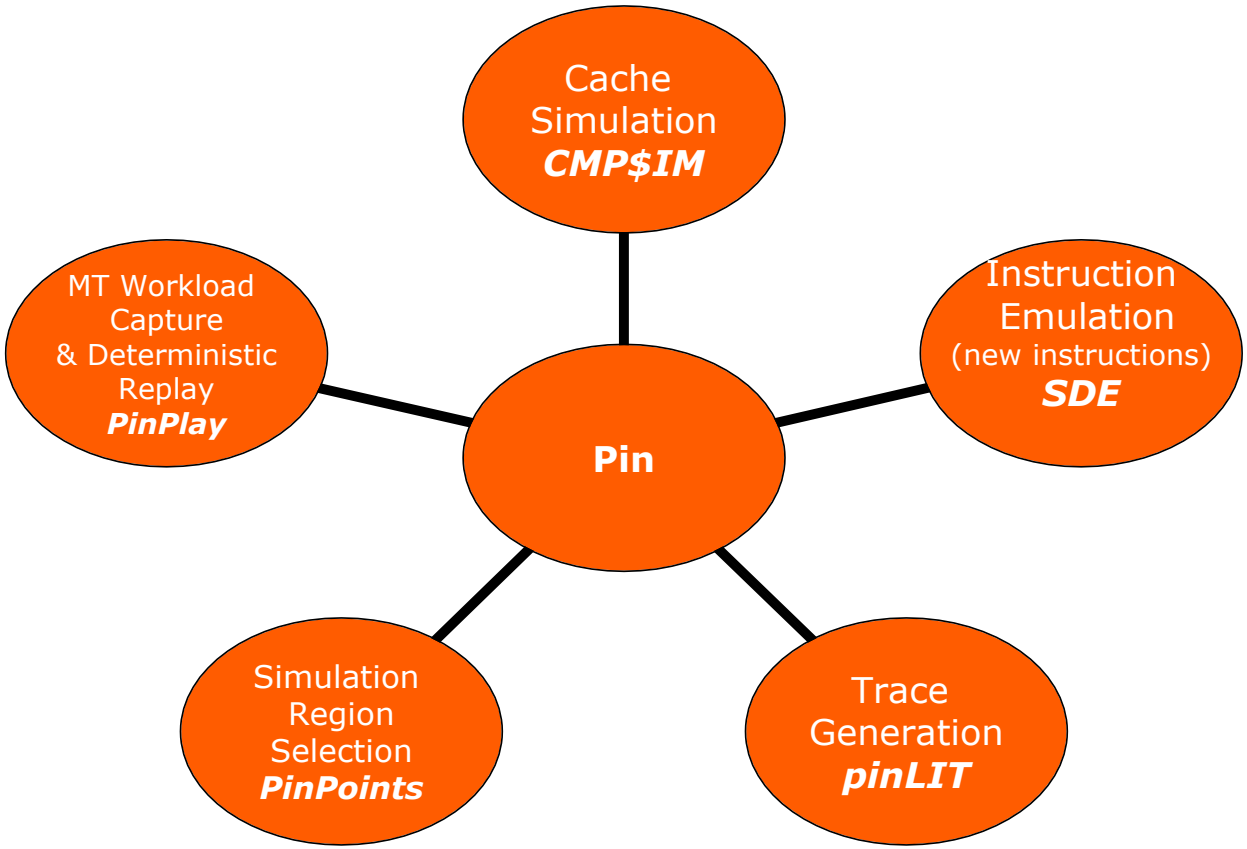
Advantages of Pin Instrumentation

- 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, Itanium
  - Supports Linux, Windows
- Robust:
  - Instruments real-life applications: Database, web browsers,...
  - Instruments multithreaded applications
  - Supports signals
- Efficient:
  - Applies compiler optimizations on instrumentation code

Pin Instrumentation Capabilities

1. Replace application functions with your own
   - Call the original application function from within your replacement function
2. 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 EIP), Register values by reference (for modification)
     - Memory address read/written by the instruction
     - Full register context
     - ...
3. Track function calls including syscalls and examine/change arguments
4. Track application threads
5. Intercept signals
6. Instrument a process tree
7. Many other capabilities ...

Example Pin-tools



Using Pin

- Launch and instrument an application:

```
$pin -t pintool.so  - - application
```

1. instrumentation engine (provided)
2. instrumentation tool (write your own, or use a provided sample)

- Attach to a running process, and instrument application:

```
$pin -t pintool.so  -pid 1234
```

Pin Instrumentation APIs

- Basic APIs are architecture independent:
  - Provide common functionalities like determining:
    - Control-flow changes
    - Memory accesses
- Architecture-specific APIs
  - e.g., Info about opcodes and operands
- Call-based APIs:
  - Instrumentation routines
  - Analysis routines

Instrumentation vs. Analysis

- Instrumentation routines define where instrumentation is inserted
  - e.g., before instruction
  - Occurs first time an instruction is executed
- Analysis routines define what to do when instrumentation is activated
  - e.g., increment counter
  - Occurs every time an instruction is executed

ManualExamples/itrace.cpp

```
#include <stdio.h>
%#include "pin.h"
FILE * trace;
 void printip(void *ip) { fprintf(trace, "%p\n", ip); }

 void Instruction(INS ins, void *v) {
    INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)printip, IARG_INST_PTR, IARG_END);
 }
void Fini(INT32 code, void *v) { fclose(trace); }
int main(int argc, char * argv[]) {
    trace = fopen("itrace.out", "w");
    PIN_Init(argc, argv);
     INS_AddInstrumentFunction(Instruction, 0);

    PIN_AddFiniFunction(Fini, 0);
    PIN_StartProgram();
    return 0;
}
```

Examples of Arguments to Analysis Routine

- IARG_INST_PTR
  - Instruction pointer (program counter) value
- IARG_UINT32 ¡value¿
  - An integer value
- IARG_REG_VALUE ¡register name¿
  - Value of the register specified
- IARG_BRANCH_TARGET_ADDR
  - Target address of the branch instrumented
- IARG_MEMORY_READ_EA
  - Effective address of a memory read
- And many more . . . (refer to the Pin manual for details)

Pintool Example: Instruction trace

- Need to pass the ip argument to the printip analysis routine

```
printip(ip)
sub         $0xff, %edx
printip(ip)
cmp         %esi, %edx
printip(ip)
jle         <L1>
printip(ip)
mov         $0x1, %edi
printip(ip)
add         $0x10, %eax
```

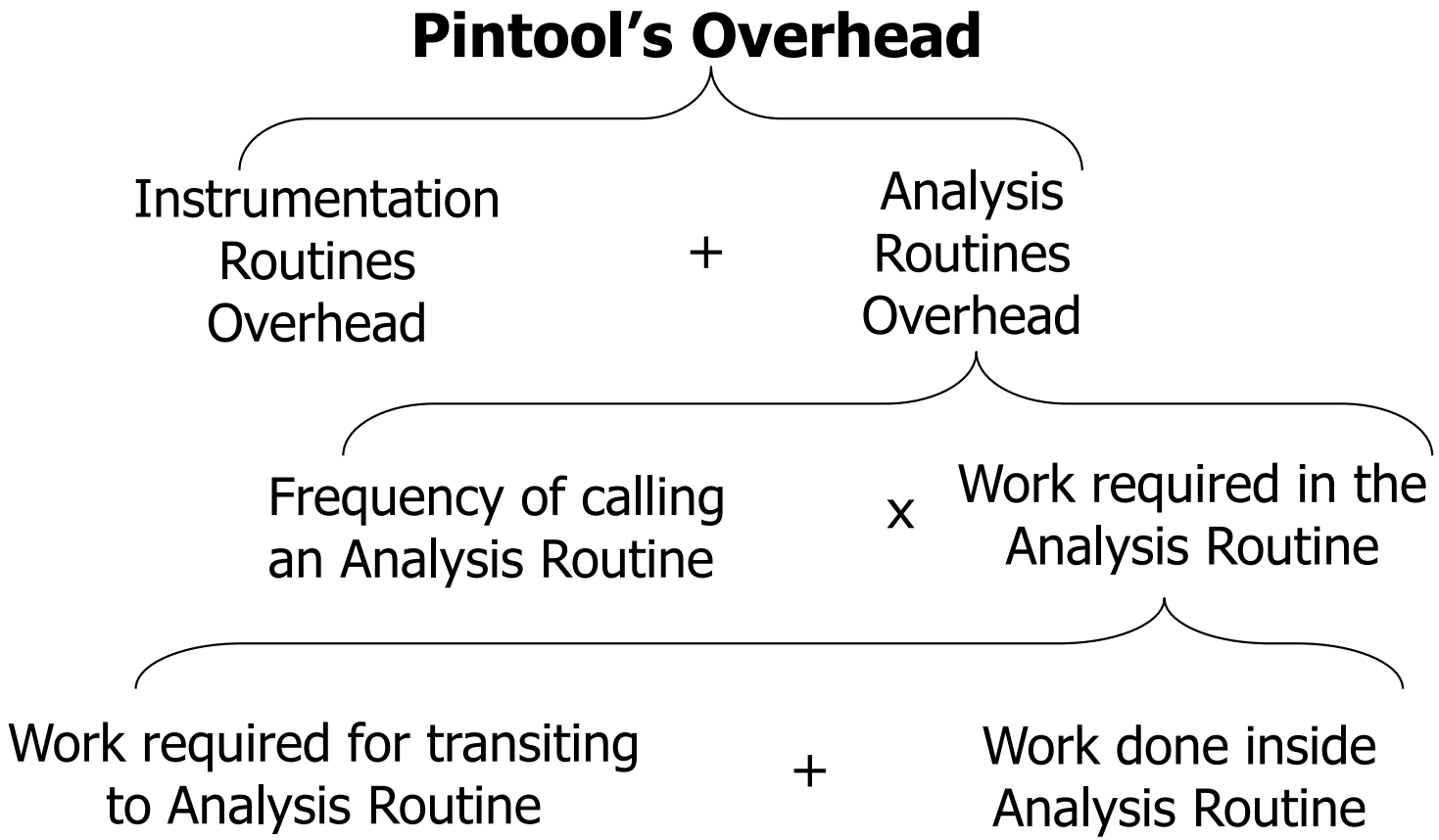
Running itrace tool

```
$ /opt/pin/pin
-t /opt/pin/source/tools/ManualExamples/
obj-intel64/itrace.so
-- /bin/ls

(...)
```

```
$ head itrace.out
0x7f907b188af0
0x7f907b188af3
0x7f907b189120
0x7f907b189121
0x7f907b189124
```

Reducing the Pintool's Overhead



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

Writing your own Pintool

- It's easier to modify one of the existing tools, and re-use the existing makefile
- Install PIN package in your home directory, and work from there
  - /opt/pin-¡version¿-¡architecture¿-¡os¿.tar.gz

Outline

Summary

Valgrind

Dyninst

References

- http://en.wikipedia.org/wiki/QEMU
- http://wiki.qemu.org/Main_Page
- http://valgrind.org/
- http://www.pintool.org/
- http://www.dyninst.org/