Lecture 5:
Dynamic Binary Instrumentation

Sanchuan Chen

schen@auburn.edu

8/31/2023

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**.

instrumentation.c

```
#include<stdio.h>
#include<string.h>

int main(int argc, char **argv){
    if(argc==1) {
        printf("Please provide the password %s\n", argv[0]);
        return 0;
    }
    if(argc>2) {
        printf("Please provide just the password, no other arguments");
        return 0;
    }
    if (!strcmp("COMP6700",argv[1])) {
        printf("I am inside the true branch\n");
    }
    else {
        printf("I am inside the else branch\n");
    }
    return 0;
}
```

What Is Dynamic Binary Instrumentation (DBI)

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

- An extremely powerful technique for program analysis.
- Dynamically inserts **extra analysis code** into the running binary program to observe how it behaves.

Applications

1. Performance profiling
2. Architecture simulation
3. Program shepherding
4. Program optimization
5. Dynamic taint analysis
6. **Reverse engineering**
7. Malware analysis

What Can be Achieved with DBI?

- 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:
  - Pin, Valgrind, QEMU, DynInst, ...

Instrumentation approaches

- **Source vs Binary**
  - Instrument source programs
  - Instrument executables directly
- Advantages for binary instrumentation
  - Language independent
  - Machine-level view
  - Instrument legacy/proprietary software
- **Static vs. Dynamic**
  - 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

- 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.

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

Advantages of Pin Instrumentation

- Easy-to-use Instrumentation:
  - Uses dynamic instrumentation
    - Does not need source code, recompilation
- Programmable Instrumentation:
  - Provides rich APIs to write in C/C++ your own instrumentation tools (called Pintools)
- Multiplatform:
  - Supports x86, x86-64, Itanium, 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, Intercept signals
5. Many other capabilities ...

How to use Pin

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

- Launch and instrument an application:
  - $ pin -t pintool.so - - application
- Attach to a running process, and instrument application:
  - $ pin -t pintool.so -pid 1234

https://software.intel.com/sites/landingpage/pintool/docs/98749/Pin/doc/html/index.html

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
```

Recap of Example Pintool: instruction count

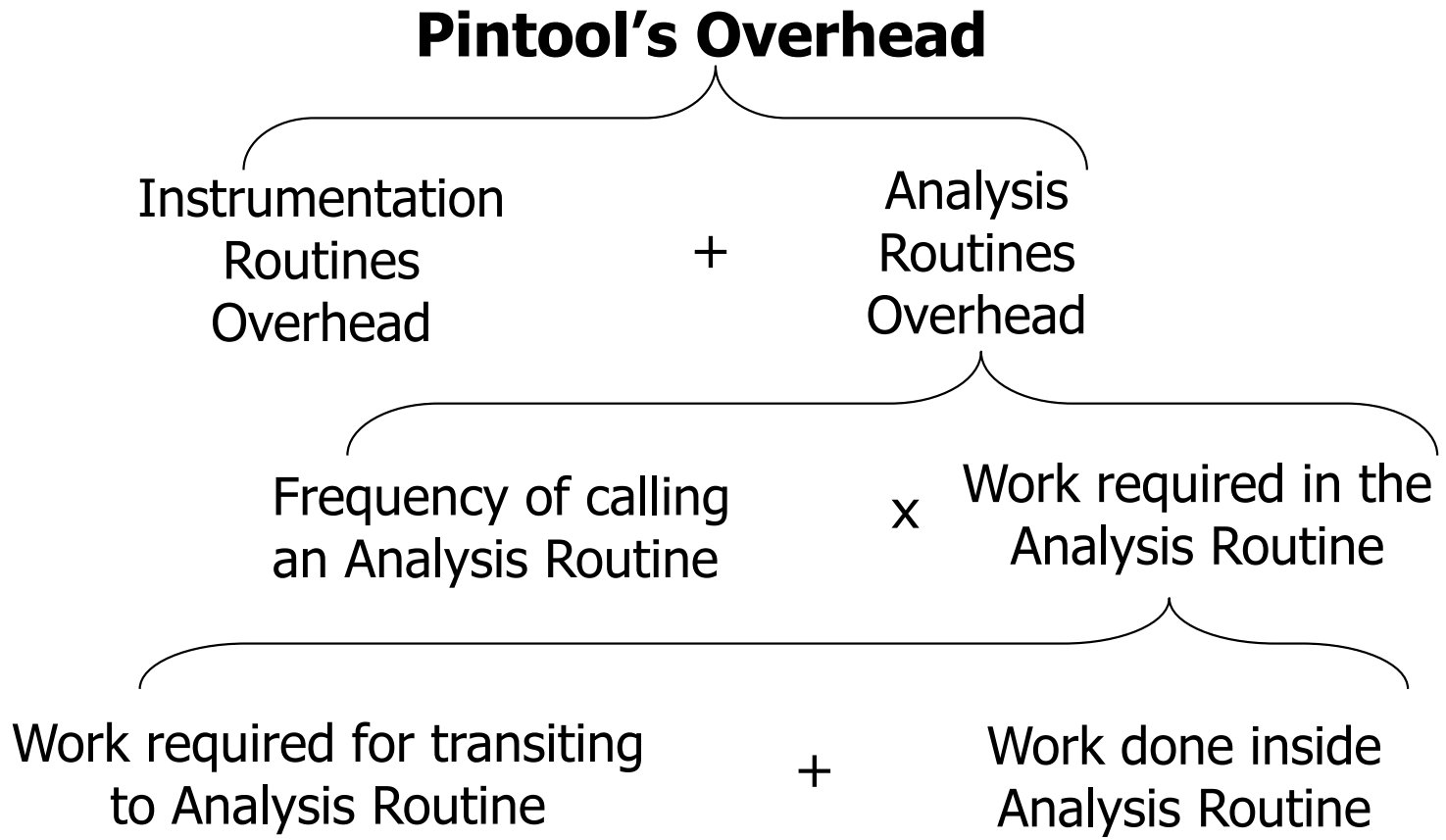
- The straightforward implementation works, but counting can be more effiicent

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

ManualExamples/inscount1.cpp

```
#include <stdio.h>
#include "pin.H"
UINT64 icount = 0;
void 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(bbl, IPOINT_BEFORE, (AFUNPTR)docount,
          IARG_UINT32, BBL_NumIns(bbl), IARG_END);
  }
}
void Fini(INT32 code, void *v) {
  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;
}
```

Reducing the Pintool's Overhead



Valgrind

- Valgrind is an instrumentation framework for building dynamic analysis tools.
- Developed by Julian Seward at/around Cambridge University,UK
  - Google-O'Reilly Open Source Award for "Best Toolmaker" 2006
  - A merit (bronze) Open Source Award 2004
  - Open source
    - works on x86, AMD64, PPC code

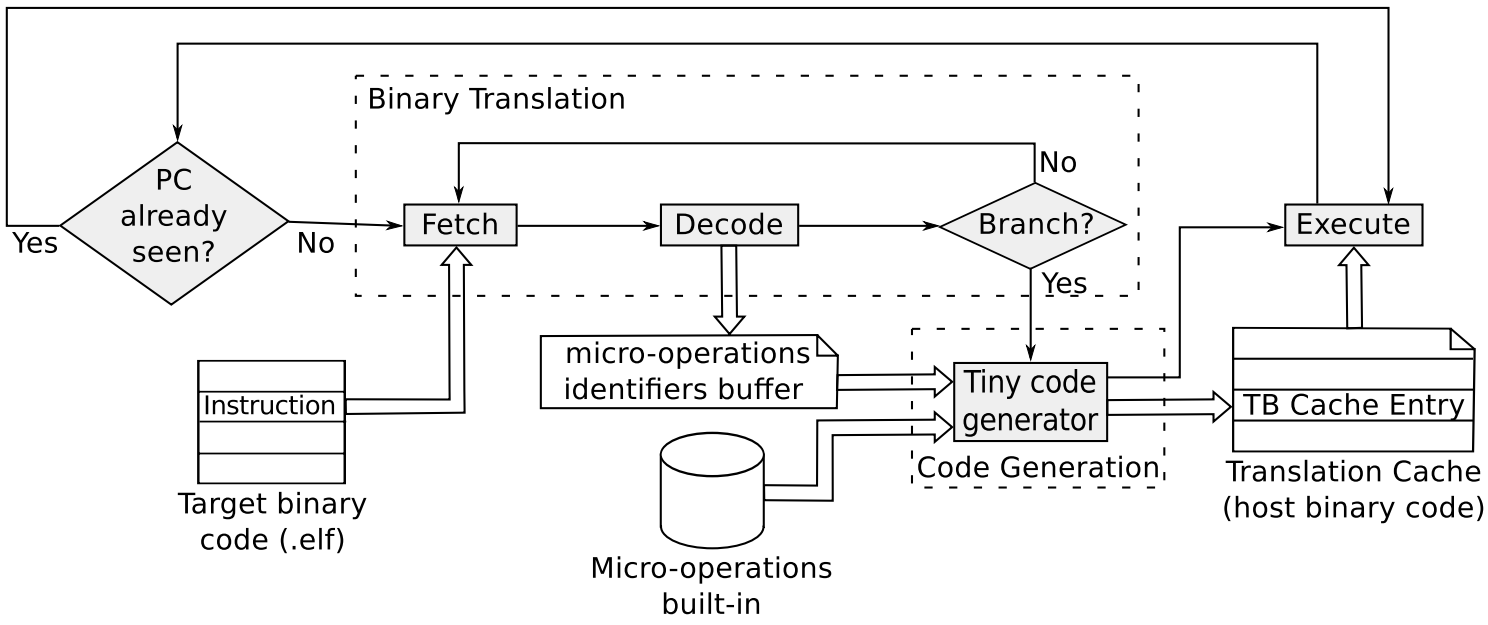
Dynamic Instrumentation - Valgrind

- Easy to execute, e.g.:
  - valgrind –tool=memcheck ls
- It becomes very popular
  - One of the two most popular dynamic instrumentation tools
    - Pin and Valgrind
  - Very good usability, extendable, robust
    - 1.09 Millions Lines of Code
- Overhead is the problem
  - 5-10X slowdown without any instrumentation

QEMU

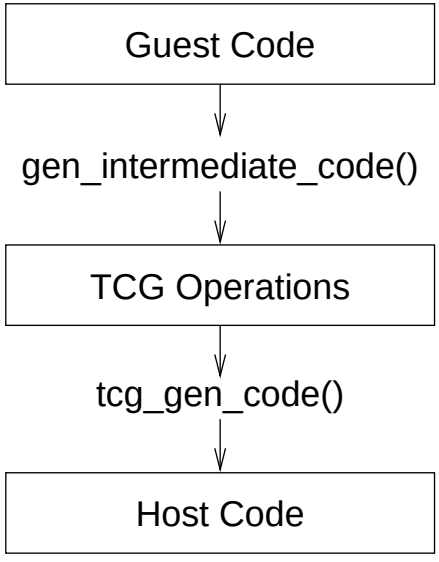
- QEMU is a generic and open source **machine emulator** and **virtualizer**.
- As a machine **emulator**, QEMU can run OSes and programs made for one machine (e.g. an ARM board) on a different machine (e.g. your own PC). By using dynamic translation, it achieves very good performance.
- As a **virtualizer**, 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

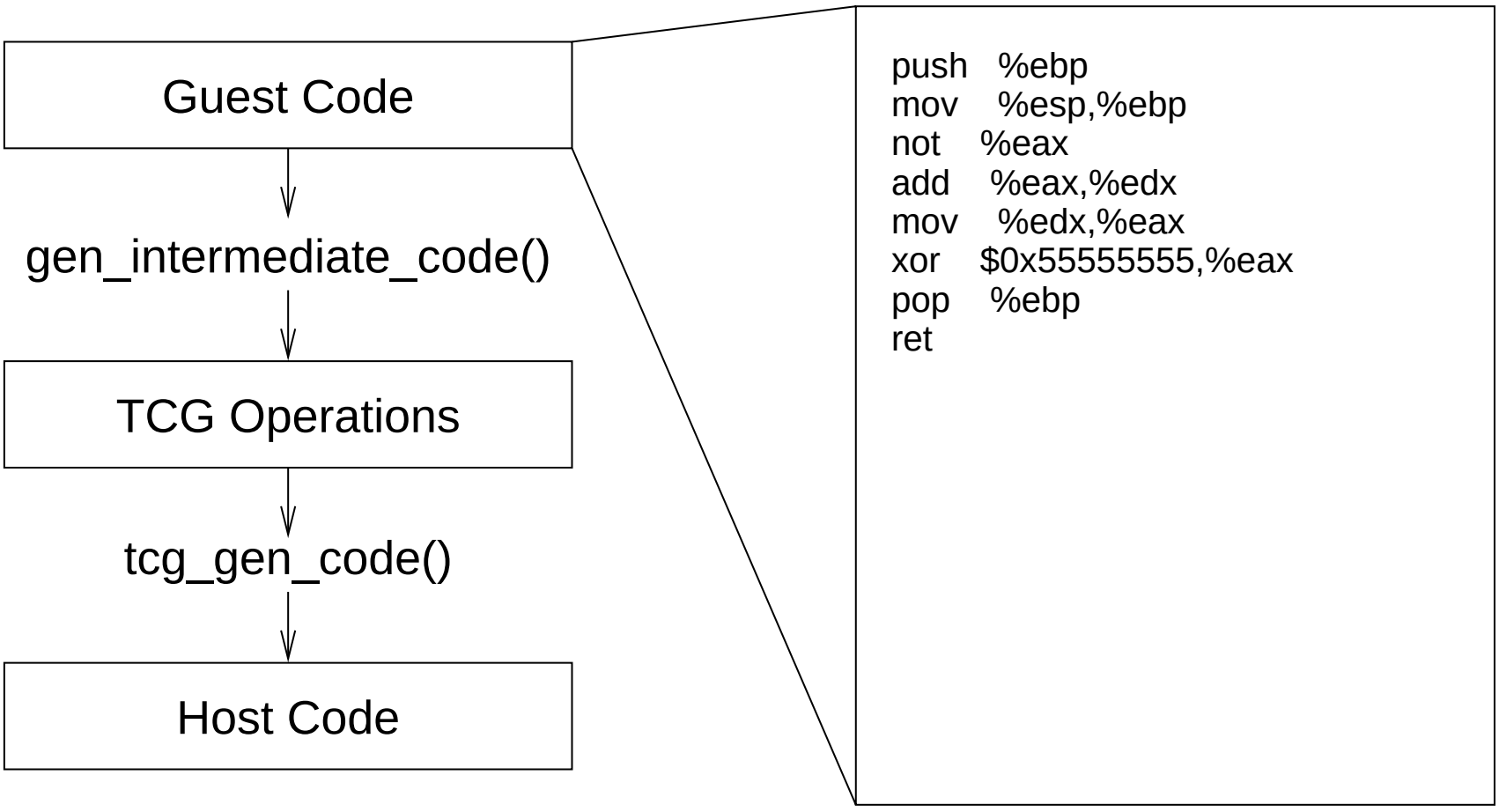


QEMU-Code Translation

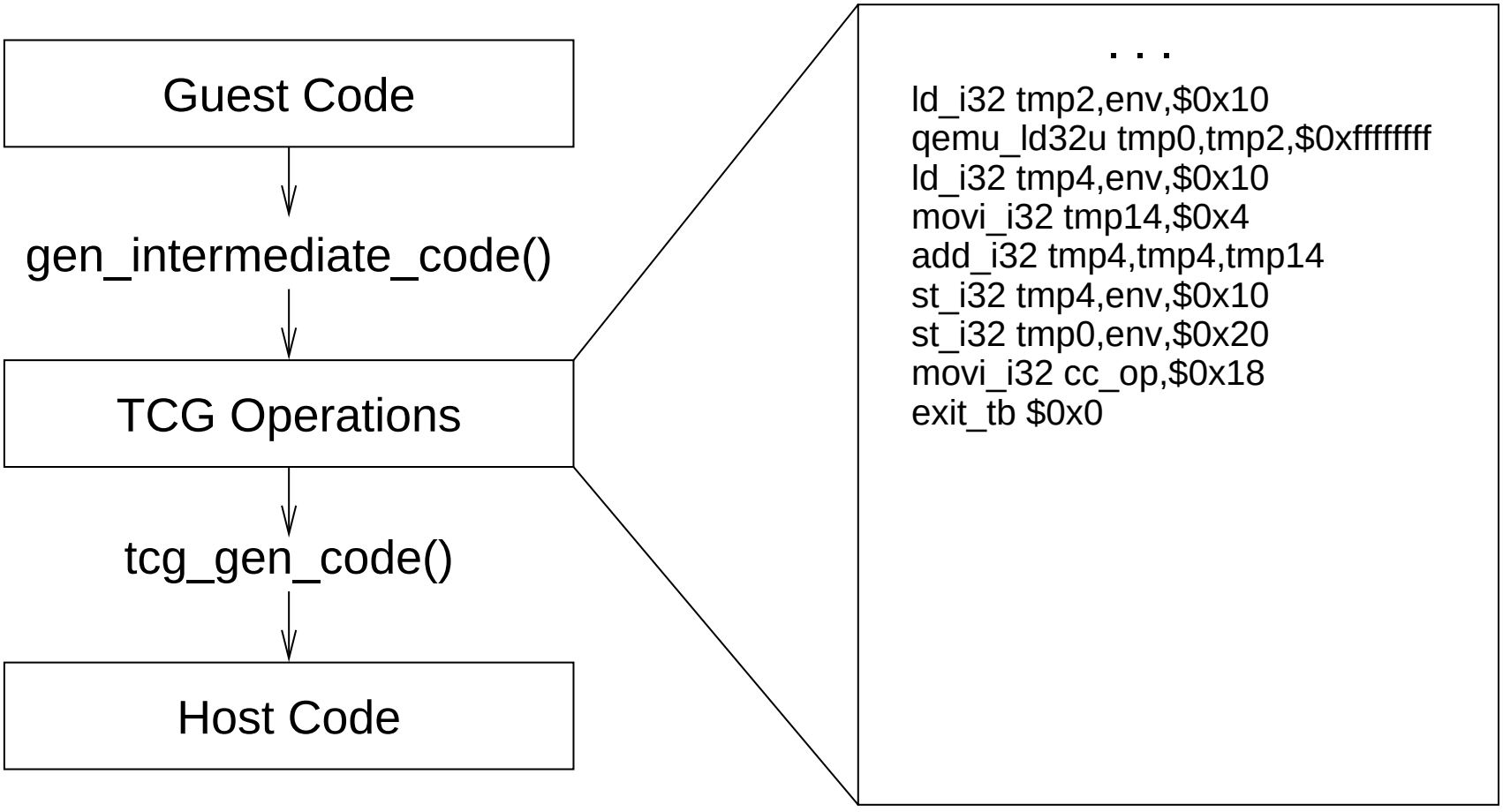
- QEMU uses an intermediate form.
- Frontends are in target-*/, includes alpha, arm, cris, i386, m68k,mips, ppc, sparc, etc.
- Backends are in tcg/*, includes arm/, hppa/, i386/, ia64/, mips/, ppc/, ppc64/, s390/, sparc/, tcg.c, tcg.h, tcg-opc.h, tcg-op.h, tcg-runtime.h



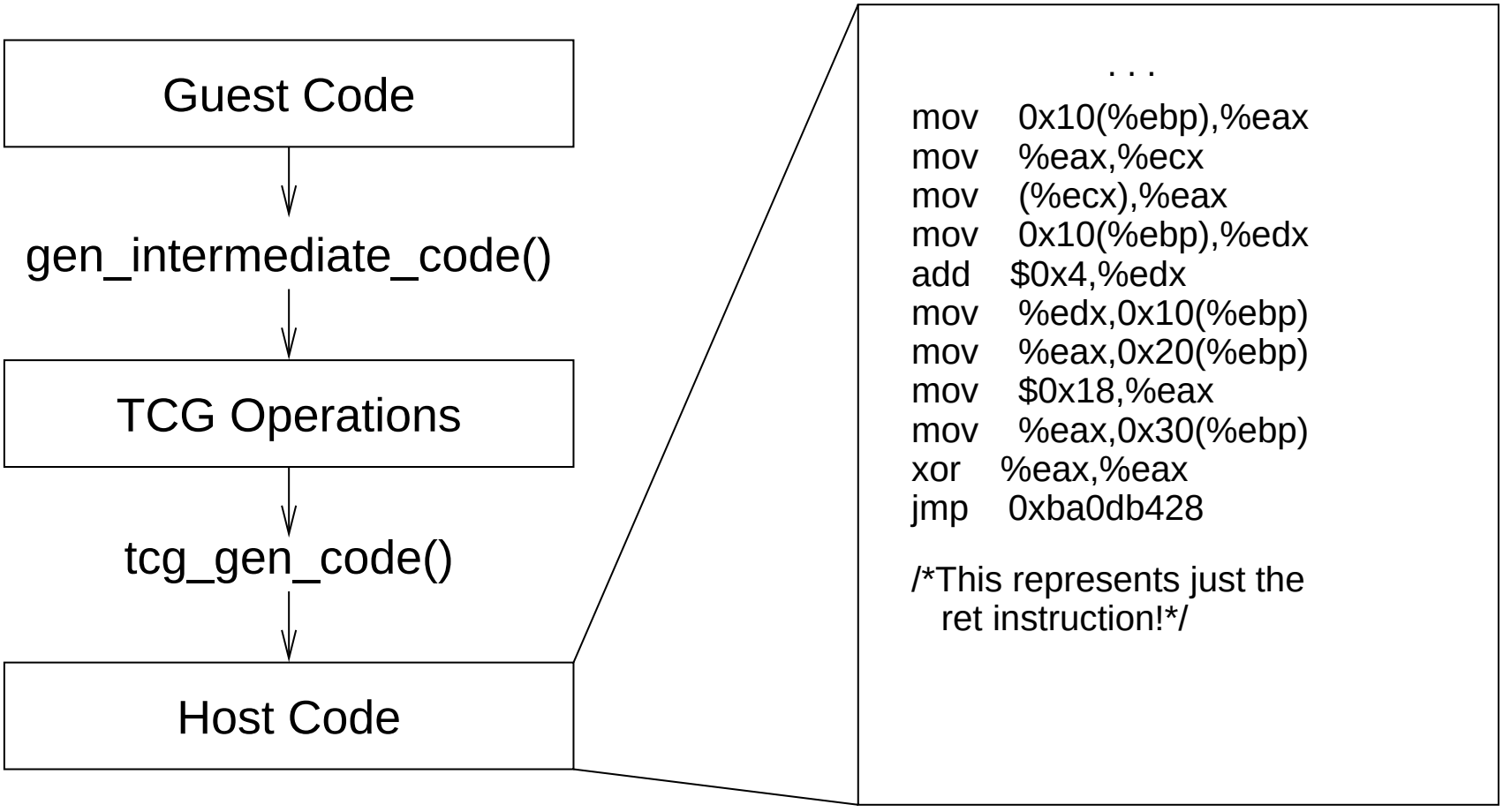
QEMU-Code Translation



QEMU-Code Translation



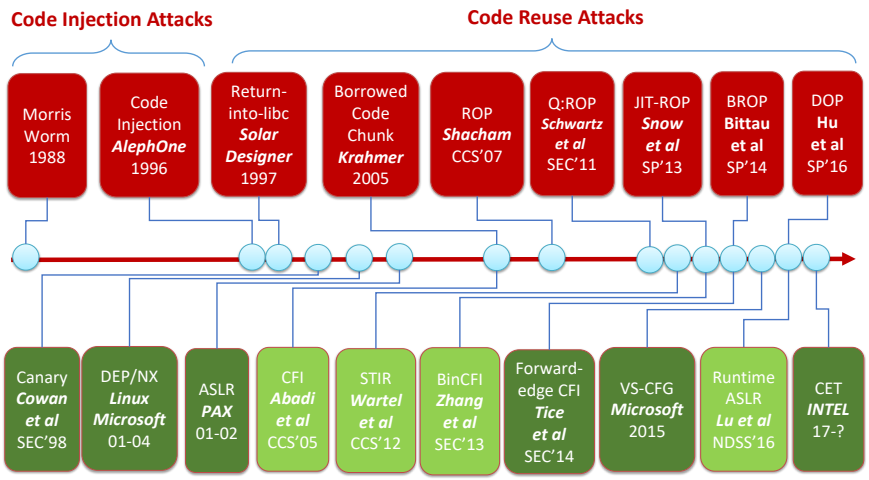
QEMU-Code Translation



QEMU use cases

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

Thank You



[1]

schen@auburn.edu
schuan.github.io

[1] Instructor appreciates the help from Prof. Zhiqiang Lin.