Advanced Topics in Malware Analysis

Execution Tracing

Brendan Saltaformaggio, PhD
Assistant Professor
School of Electrical and Computer Engineering

Tracing: Source Level Instrumentation

1

Lesson Objectives

- Employ the execution tracing for dynamic analysis
- Apply static and dynamic binary instrumentation
- Create a Pintool to utilize PIN
- Discuss Valgrind and QEMU techniques for instrumentation

2

The Dynamic Analysis Swiss Army Knife

- Tracing is a process that faithfully records detailed information of program execution
  - **Benefits**: Lossless, simple to implement, low overhead
  - **Problems**: Requires heavy after-the-fact analysis on the execution trace
- Control flow tracing
  - Logs the sequence of **executed** statements
- Dependence tracing
  - Logs the sequence of **exercised** dependencies
- Value tracing
  - Logs the sequence of **values** that are produced by each instruction
- Memory access tracing
  - Logs the sequence of **memory** references during an execution

3

You Have Already Performed Tracing!

Tracing by **printf**

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

This is the essence of dynamic tracing!

May seem silly, but…

Consider this output:

```
In the loop!
In the loop!
True branch.
In the loop!
True branch.
In the loop!
True branch.
```

- How many elements in the list?
- How many negative values?

4

Tracing by Source Level Instrumentation

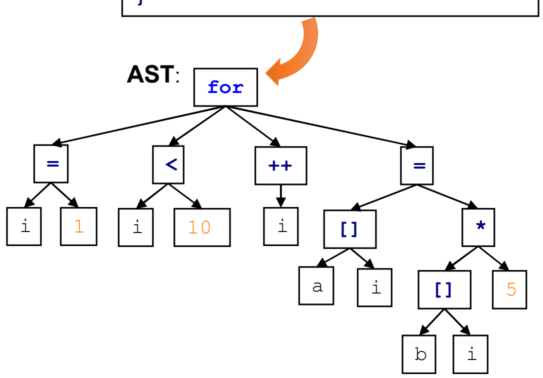
- Typically performed via a compiler plugin
  - LLVM, GCC plugin, etc.

1. Read a source file and parse it into **abstract syntax trees** (**ASTs**)
2. Annotate the ASTs with instrumentation
3. Translate the instrumented ASTs into a new source file
4. Compile the new source
5. Execute the instrumented program and a trace is produced

5

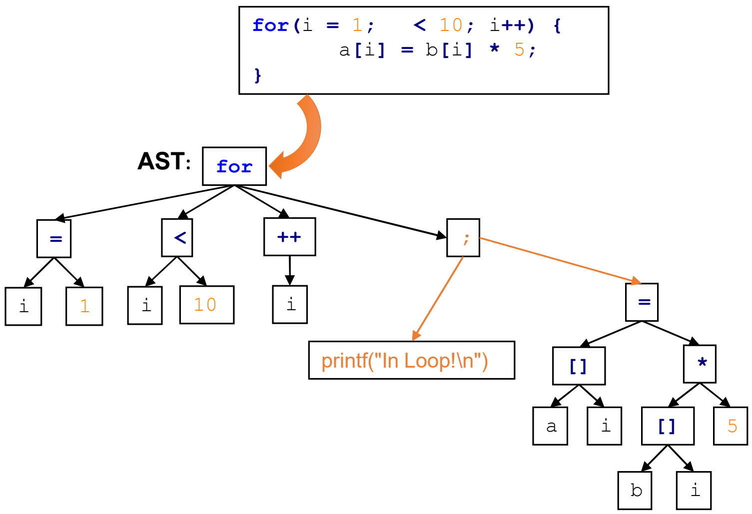
Tracing by Source Level Instrumentation

```
for(i = 1;   < 10; i++) {
        a[i] = b[i] * 5;
}
```



6

Tracing by Source Level Instrumentation



7

Limitations of Source Level Instrumentation

- **Requires source code**
  - Worms and viruses are rarely provided with source code
  - Closed source software
  - That is why we are in this class ☺
- **Hard to handle libraries**
  - Proprietary libraries: communication (MPI, PVM), linear algebra (NGA), database query (SQL libraries)
- **Hard to handle multi-lingual programs**
  - Source code level instrumentation is language dependent
  - Have to rewrite everything for Java, C++, Python, …

8

Advanced Topics in Malware Analysis
Execution Tracing

Brendan Saltaformaggio, PhD
Assistant Professor
School of Electrical and Computer Engineering

Tracing: Binary Instrumentation

Georgia Tech

9

Tracing by Binary Instrumentation

- The **most important** binary-analysis capability to date: **Binary Instrumentation**
- **Two flavors**:
    1. Static Binary Instrumentation
    2. Dynamic Instrumentation
- **Features:**
    - No source code required
    - Directly handles library binaries (either statically instrumenting or during their execution)
    - Does not care what language(s) the program is written in --- any binary can be instrumented
        - **Worst case example**: Instrument the JVM & monitor the Java program during execution

Georgia Tech

10

Static Binary Instrumentation

- Insert instrumentation into the binary executable
- Instrumentation will run next time the binary is executed
- A.K.A. Binary "Rewriting"

1. Given a binary executable, **parse** it into **intermediate** representation
   - Could be as simple as instruction models (like we did in Lab 3 & Lab 4)
   - More advanced representations such as control flow graphs may also be generated
2. Design tracing instrumentation for the intermediate representation
   - E.g., Control flow tracing → for each (instruction in binary): printf("%i", instruction.address);
3. A lightweight "compiler" inserts the instrumentation logic into a new executable

Static Binary Instrumentation Example

Original Source:

```c
#include <stdio.h>

int main(int argc, char* argv[]) {
  if (argc == 2)
    printf("Hello %s\n", argv[1]);
  return 0;
}
```

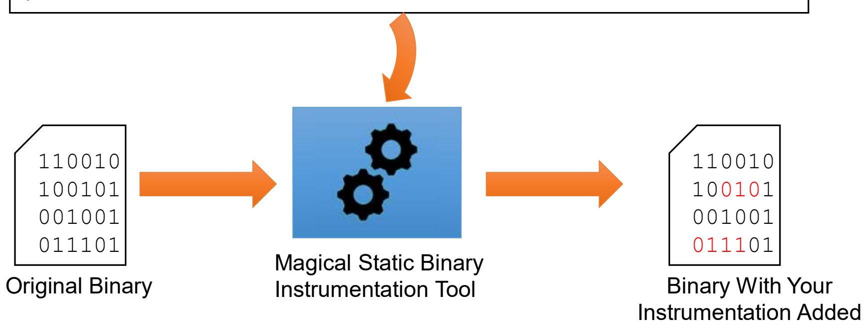
Original Binary
(64-bit Linux with -O3)

```
        cmp     edi, 2
        jz      print
        xor     eax, eax
        retn
print:  push    rax
        mov     rsi, [rsi+8]
        mov     edi, offset format ; "Hello %s\n"
        xor     eax, eax
        call    _printf
        xor     eax, eax
        pop     rdx
        retn
```

Static Binary Instrumentation Example

Your Instrumentation Logic:

```
void trace_ctrl_flow(inst *current_inst) {
    printf("%u\n", current_inst->address);
}
...
for(inst *in = get_instructions(); in; in = in->next) {
    add_instrumentation(in, trace_ctrl_flow);
}
```



Georgia Tech

13

Static Binary Instrumentation Example

```
inst:      push   rsi    ; save all dirty regs
           push   rax
           push   rflags
           mov    rsi, rdi  ; embedded instrumentation
           mov    edi, offset u ; "%u\n"
           call   _printf
           pop    rflags ; restore dirty regs
           pop    rax
           pop    rsi
           ret
...
1.         cmp       edi, 2
           push   rdi    ; call instrumentation
           mov    rdi, 1
           call   inst
           pop    rdi
           push   rdi     ; call instrumentation
           mov    rdi, 2
           call   inst
           pop    rdi
2.         jz        print   ; be careful on ctrl flow!
```

14

Static Binary Instrumentation Example

```
3.          xor     eax, eax
            push    rdi    ; call instrumentation
            mov     rdi, 3
            call    inst
            pop     rdi
            push    rdi    ; call instrumentation
            mov     rdi, 4
            call    inst
            pop     rdi
4.          retn
5. print:   push    rax
            push    rdi    ; call instrumentation
            mov     rdi, 5
            call    inst
            pop     rdi
6.          mov     rsi, [rsi+8]
            push    rdi    ; call instrumentation
            mov     rdi, 6
            call    inst
            pop     rdi
```

15

The Good & Bad of Static Binary Instrumentation

- Almost impossible to do accurately!
  - Pointers must be carefully preserved or adjusted
  - Memory data must not be corrupted
  - Original disassembly must be 100% correct
    - You cannot instrument data that was misinterpreted as code!
  - Suggested readings:
    - Deng, Z., Zhang, X., & Xu, D. (2013). BISTRO: Binary Component Extraction and Embedding for Software Security Applications. *Proceedings of the 18th European Symposium on Research in Computer Security- – ESORICS 2013*.
    - Wartell, R., Mohan, V., Hamlen, K. W., & Lin, Z. (2012). Securing untrusted code via compiler-agnostic binary rewriting. *Proceedings of the 28th Annual Computer Security Applications Conference- ACSAC 12.*
- Significantly increases executable file size
- Good: FAST! All the instrumentation logic is “baked in” to the executable

16

Binary Instrumentation: Static vs Dynamic

- **Static Binary Instrumentation:** Given an original binary executable and generate an instrumented executable that can be executed with our analysis embedded within!
  - Instrument statically = before runtime
- **Dynamic Binary Instrumentation:** Given an original binary executable and an input, start executing the binary with the input, and during execution add instrumentation to the binary on the fly
  - 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

17

Dynamic Binary Instrumentation Example

- Original Source:

```c
#include <stdio.h>

int main(int argc, char* argv[]) {
  if (argc == 2)
    printf("Hello %s\n", argv[1]);
  return 0;
}
```

Original Binary
(64-bit Linux with -O3)

```
           cmp     edi, 2
           jz      print
           xor     eax, eax
           retn
print:     push    rax
           mov     rsi, [rsi+8]
           mov     edi, offset format ; "Hello %s\n"
           xor     eax, eax
           call    _printf
           xor     eax, eax
           pop     rdx
           retn
```

18

Dynamic Binary Instrumentation Example

Your Instrumentation Logic:

```
void trace_ctrl_flow(inst *current_inst) {
    printf("%u\n", current_inst->address);
}
...
for(inst *in = get_instructions(); in; in = in->next) {
    add_instrumentation(in, INST_AFTER, trace_ctrl_flow);
}
```

Notice: Could also be INST_BEFORE or others…

```
cmp     edi, 2
jz      print
push    rax
mov     rsi, [rsi+8]
```

Instructions that should be executed (starting from entry point)

Magical Dynamic Instrumentation Tool

```
cmp     edi, 2
push    rdi; call inst
mov     rdi, 1
call    inst
pop     rdi
```

Instruction with added implementation that will be executed

Georgia Tech

19

Dynamic Binary Instrumentation

- A few flavors.
- We will look at:
  - **Platform-specific** --- instrumentation targets the exact instructions that are executing on the CPU
  - **Emulation-based** --- instrumentation targets some intermediate language
  - Others (some are hybrids of these categories)

Pin

QEMU

Valgrind

DynamoRIO The DR. is in.

Dyninst

Georgia Tech

20

Advanced Topics in Malware Analysis

Execution Tracing

Brendan Saltaformaggio, PhD

Assistant Professor

School of Electrical and Computer Engineering

PIN

Georgia Tech

21

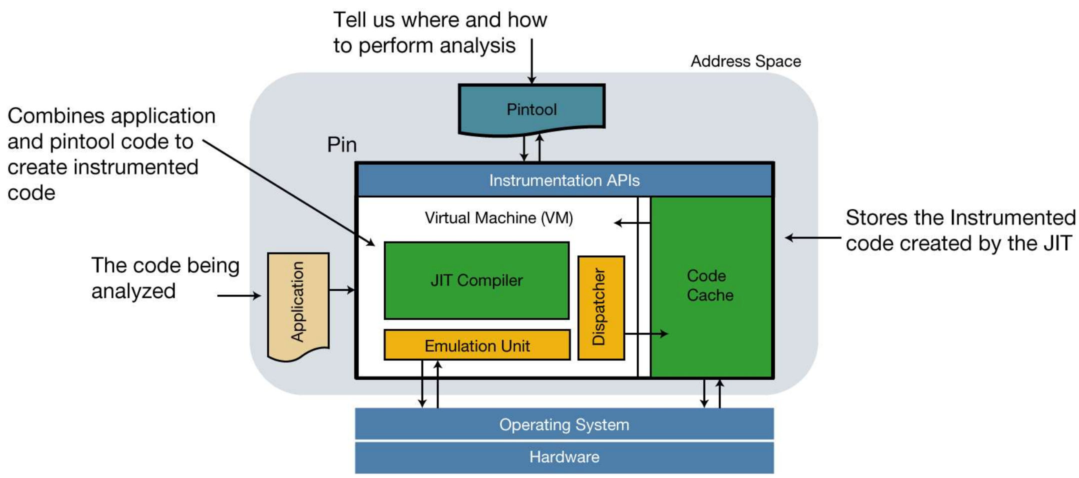
The King of x86 Dynamic Instrumentation: PIN

- **Pin** is a platform for dynamic instrumentation of binary programs
- **Pin** is maintained by **Intel**
- Supports Intel architectures: **x86, x86-64**, **Itanium**, **Xscale**
- Can instrument executables on a variety of OS's: Windows, Linux, OSX, Android
- You write a "**Pintool**" which instructs Pin on how you want to instrument the execution
- Pintools are very powerful (that is why Pin is king) because they can **insert arbitrary code** (written in C or C++) at **arbitrary locations** in an executable
- Instrumentation code is added dynamically while the executable is running
  - It can also be changed on-the-fly!
  - Handles threads & asynchronous signals!
  - This makes it possible to attach Pin to an already running process

Georgia Tech

22

The King of x86 Dynamic Instrumentation: PIN



Read More: Pin: building customized program analysis tools with dynamic instrumentation. In ACM SIGPLAN Notices (Vol. 40, No. 6, pp. 190-200).

Georgia Tech

23

PinTools

- https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool
- Launch and instrument an application:

$ pin -t pintool.so --application

Instrumentation engine
(provided in the kit)

Instrumentation tool(write your own, or
use one provided in the kit)

- Attach to and instrument an application: $ pin -t pintool.so -pid 1234
- Like IDA, Pintools are specific to 32-bit and 64-bit!
  - May share code (watch out for incompatibilities!), but must be compiled separately

Georgia Tech

24

Pintools API

- 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
- State change APIs
  - E.g., process create/end, interrupt received
- Call-based APIs:
  - Instrumentation routines
  - Analysis routines

25

Pintools Structure

Two types of core routines:

1. **Instrumentation routines** define where instrumentation is inserted
   - E.g., before instruction
   - Occurs first time an instruction is executed
   - Other APIs can remove instrumentation after it has been inserted

2. **Analysis routines** define what to do when instrumentation is activated
   - E.g., print instruction address, increment counter
   - Occurs every time an instruction is executed

26

Pintool Example:

ManualExamples/itrace.cpp

Georgia Tech

27

```
#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;
}
```

Instruction ptr arg!

Pintool Example: ManualExamples/itrace.cpp

Analysis Routine

Instrumentation Routine

State change Analysis Routine

Main calls the Pin APIs to register the instrumentation routines

Georgia Tech

28

Why PIN is King

- Versatility! Pin can do sooooo many things
- Sample tools in the Pin distribution:
  - See **C:\pin\source\tools** directory!!
  - Cache simulators, branch predictors, address tracer, syscall tracer, edge profiler, stride profiler
- Some tools developed and used inside Intel:
  - **Opcodemix** --- analyze code generated by compilers
  - PinPoints --- find representative regions in programs to simulate
- Companies are writing their own Pintools
- Universities use Pin in teaching and research
  - My PhD was one big journey in Pin --- https://github.com/bdsaltaformaggio/DSCRETE
  - Many others…

Georgia Tech

29

Advanced Topics in Malware Analysis

Execution Tracing

Brendan Saltaformaggio, PhD

Assistant Professor

School of Electrical and Computer Engineering

Valgrind

Georgia Tech

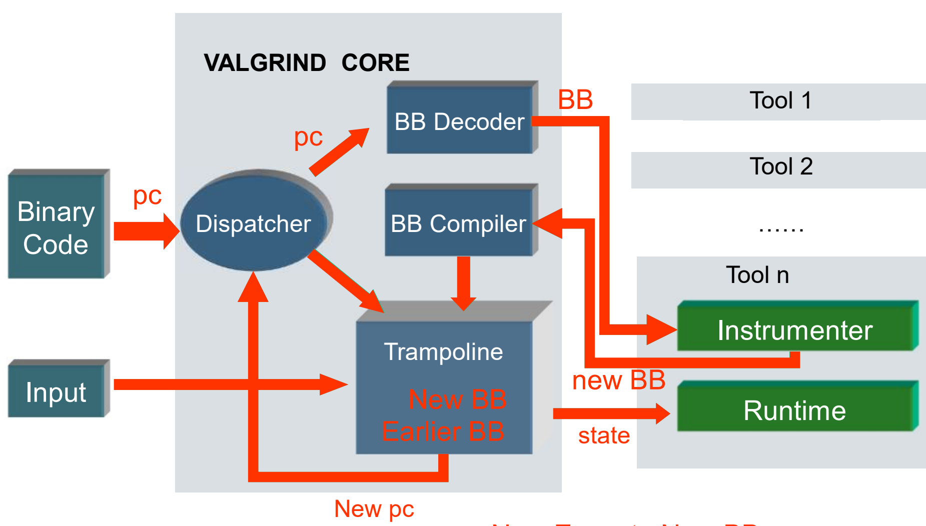
30

Dynamic Binary Instrumentation - Valgrind

- Developed by Julian Seward at Cambridge University
  - Google - O'Reilly Open Source Award for "Best Toolmaker" 2006
  - A merit (bronze) Open Source Award 2004
  - Open source (Pin is not!)
  - Works on many platforms: x86, PowerPC, ARM, …
- Overhead is a big problem
- 5-10 ***times*** slowdown **without** any instrumentation
  - Pin has very little slowdown for no instrumentation
- Suggested reading:
  - Nethercote, N., & Seward, J. (2007). Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI 07*.

31

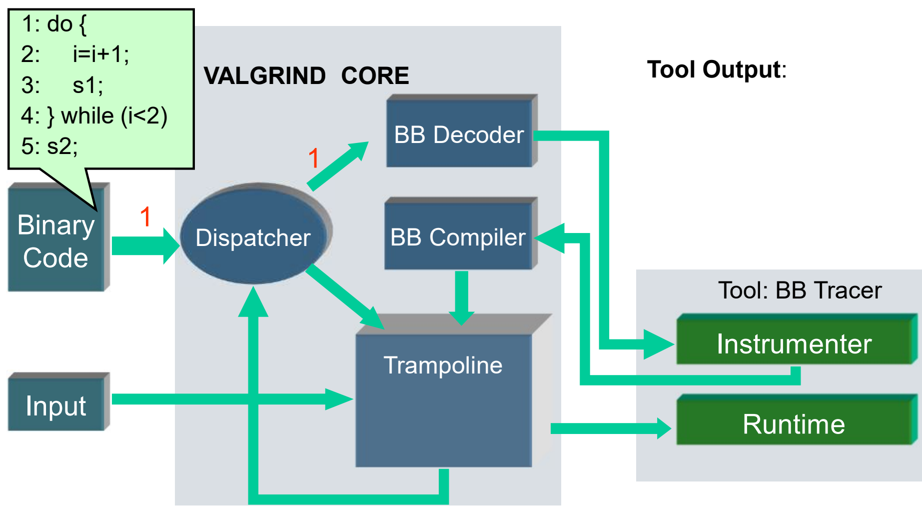
Valgrind Infrastructure



Now Execute New BB

32

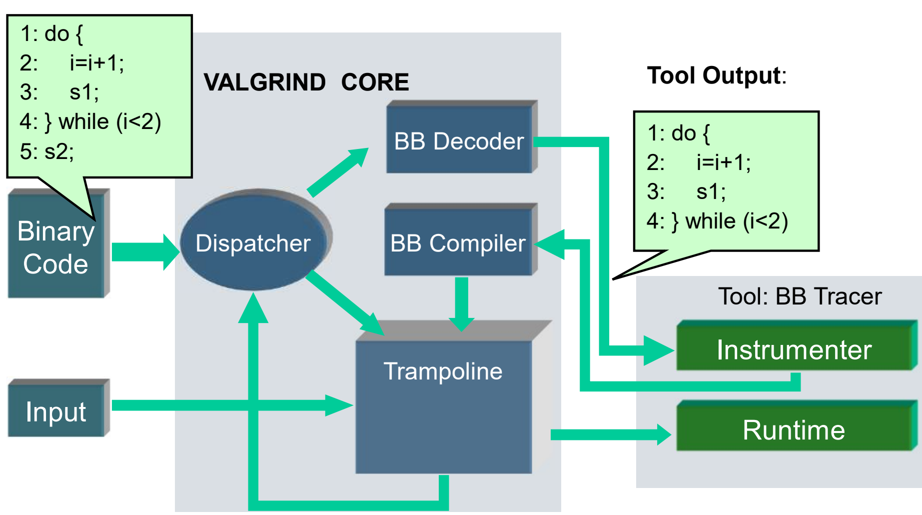
Valgrind Example



Georgia Tech

33

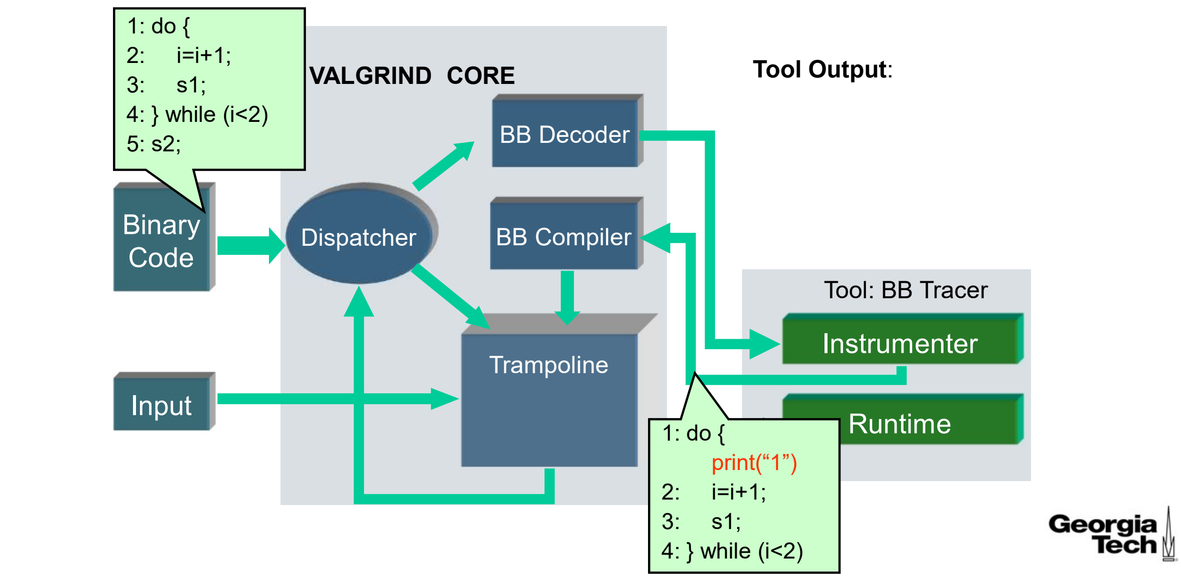
Valgrind Example



Georgia Tech

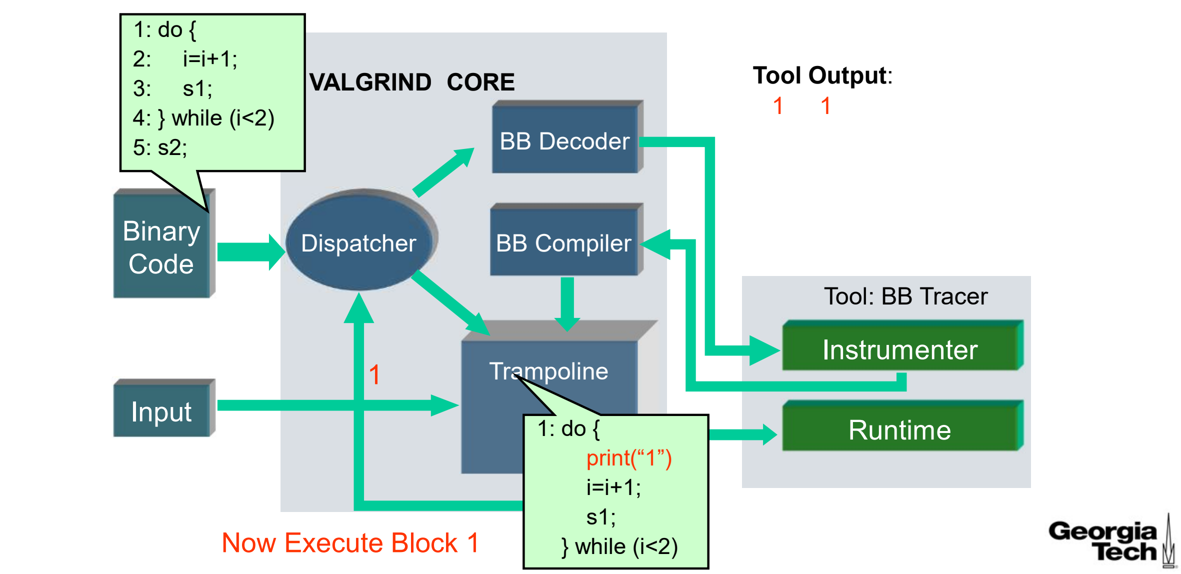
34

Valgrind Example



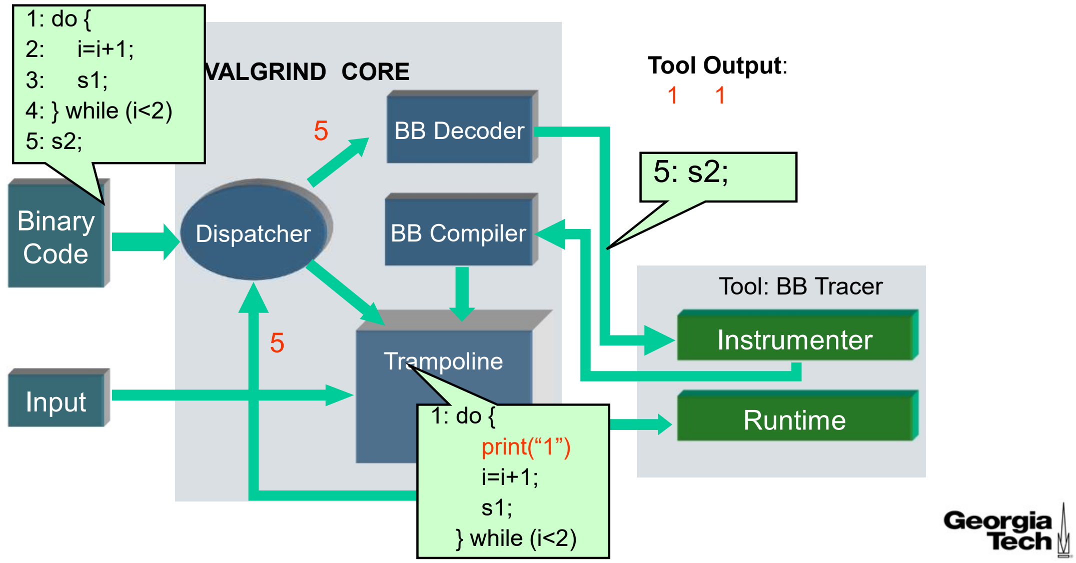
35

Valgrind Example



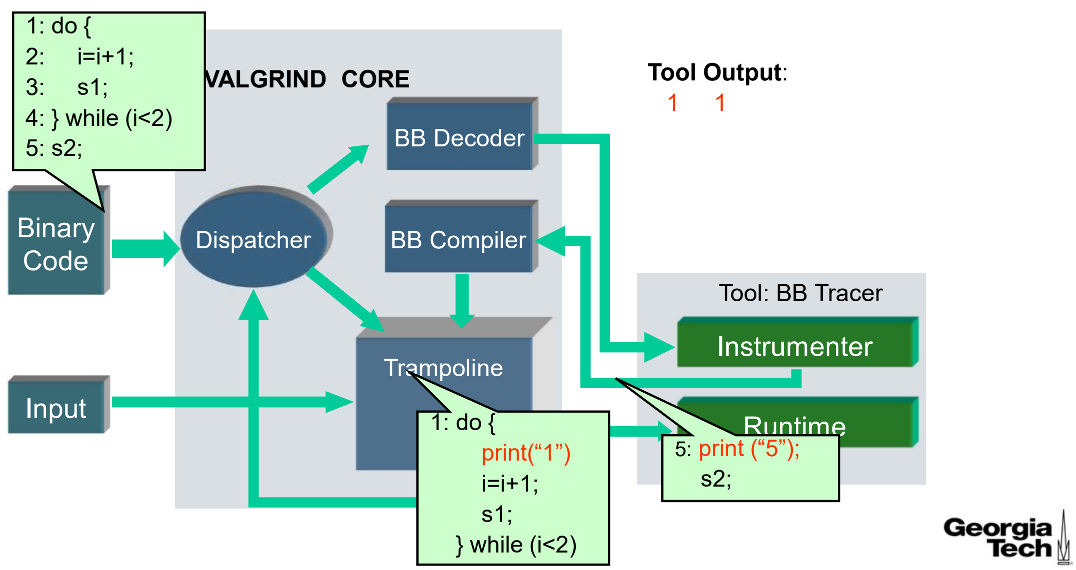
36

Valgrind Example



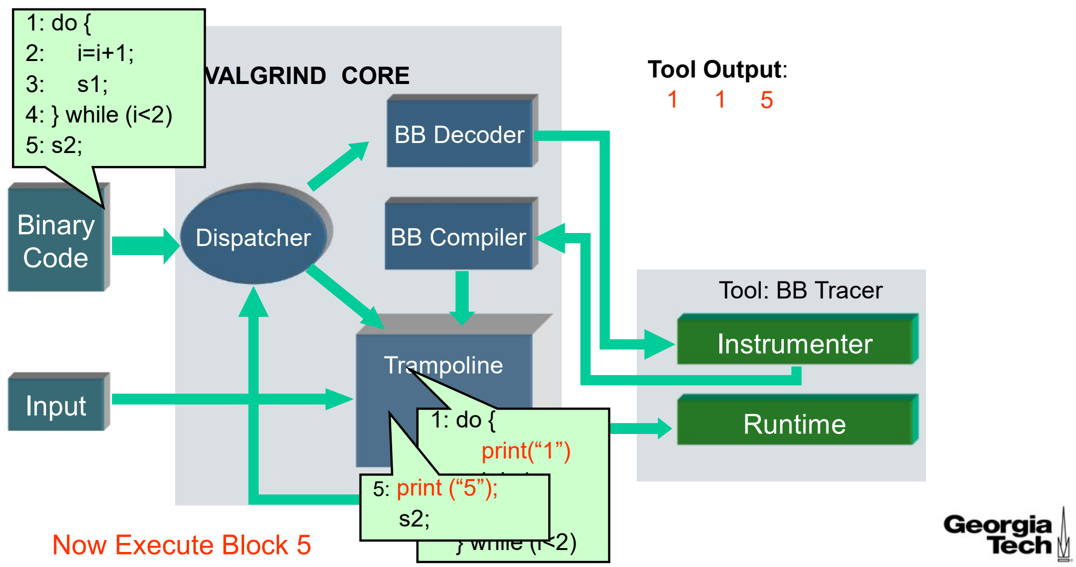
37

Valgrind Example



38

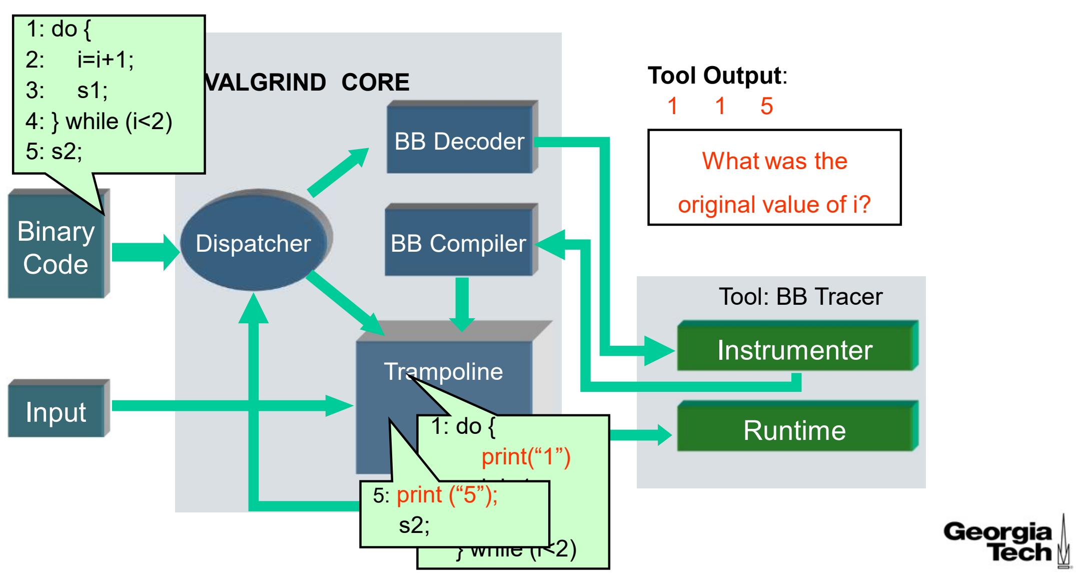
Valgrind Example



Now Execute Block 5

39

Valgrind Example



40

Advanced Topics in Malware Analysis

Execution Tracing

Brendan Saltaformaggio, PhD

Assistant Professor

School of Electrical and Computer Engineering

QEMU

Georgia Tech

41

QEMU Emulation-based Instrumentation

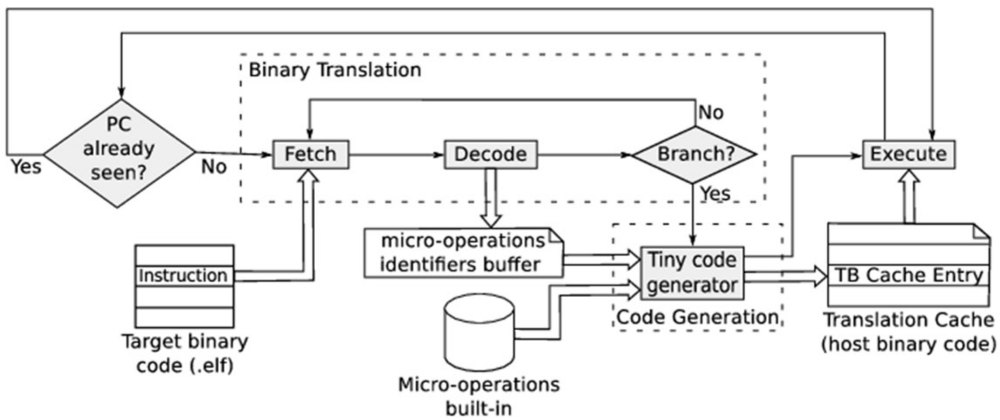
- QEMU is a generic and open source machine **emulator** and hypervisor (VM manager)
- As a **machine emulator**, QEMU can run OSes and programs made for one machine (e.g. an ARM board) on a different machine (e.g. your own PC)
  - By using dynamic translation, it achieves very good performance
- As a **hypervisor**, QEMU achieves near native performance 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
- In either case, you can add your analysis routines to QEMU's code base to instrument the entire guest OS as it is executing!

Georgia Tech

42

QEMU Infrastructure

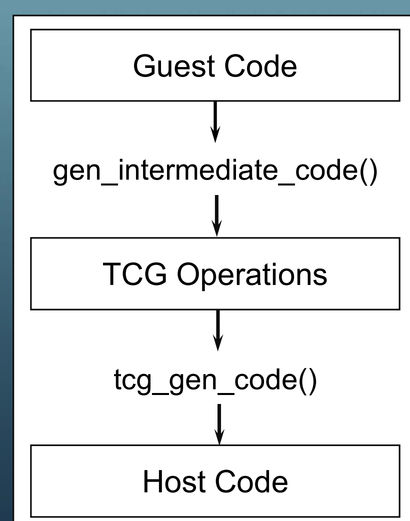
- How does QEMU execute ARM code on x86?



43

QEMU Code Translation

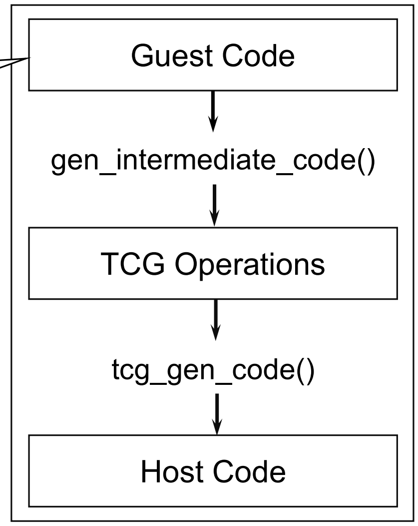
- QEMU uses an intermediate representation (IR)
- Frontends are in the “target” directory
  - These are guest architectures
  - “What QEMU executes”
  - Includes alpha, arm, cris, i386, m68k, mips, ppc, sparc, …
- Backends are in tcg/
  - “Tiny Code Generator”
  - These are host CPU operations
  - “Where QEMU executes”
  - Includes arm/, i386/, ia64/, mips/, ppc/, ppc64/, s390/, sparc/, …



44

QEMU Code Translation Example
x86 Guest -> x86 Host

```
push   ebp
mov    ebp, esp
not    eax
add    edx, eax
mov    eax, edx
xor    eax, 0x55555555
pop    ebp
ret
```

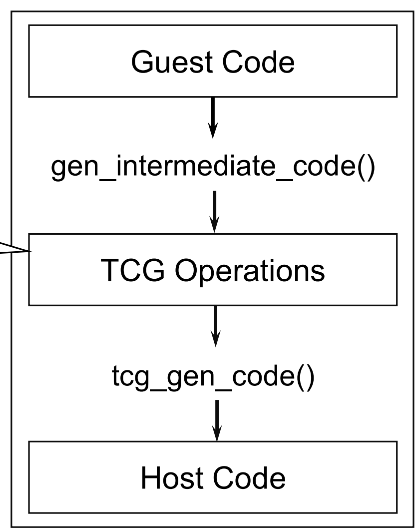


Georgia Tech

45

QEMU Code Translation Example
x86 Guest -> x86 Host

```
ld_i32        tmp2,env,$0x10
qemu_ld32u    tmp0,tmp2,$0xffffffff
ld_i32        tmp4,env,$0x10
movi_i32      tmp14,$0x4
add_i32       tmp4,tmp4,tmp14
st_i32        tmp4,env,$0x10
st_i32        tmp0,env,$0x20
movi_i32      cc_op,$0x18
exit_tb       $0x0
```

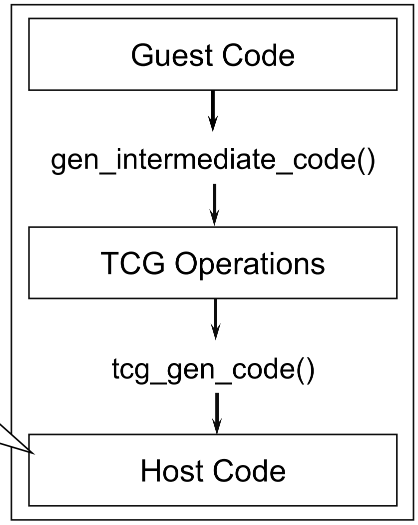


Georgia Tech

46

QEMU Code Translation Example
x86 Guest → x86 Host

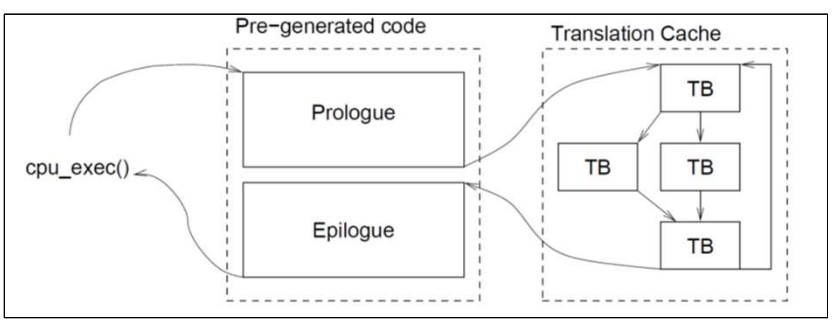
```
mov 0x10(%ebp),%eax
mov 0x10(%ebp),%edx
mov (%ecx),%eax
mov %eax,%ecx
add $0x4,%edx
mov %edx,0x10(%ebp)
mov %eax,0x20(%ebp)
mov $0x18,%eax
mov %eax,0x30(%ebp)
xor %eax,%eax
jmp 0xba0db428
. . .
/* This represents just
the ret instruction! */
```



47

Qemu Block Chaining

- Like we saw in Valgrind, returning to the dispatcher from the code cache is very slow!
- Solution: jump directly between basic blocks!
  - Make space for a jump, followed by a return to the epilogue
  - Every time a block returns, try to chain it



48

Advanced Topics in Malware Analysis
Execution Tracing

Brendan Saltaformaggio, PhD
Assistant Professor
School of Electrical and Computer Engineering

Offline Trace Reconstruction

Georgia Tech

49

Key Word: Lossless

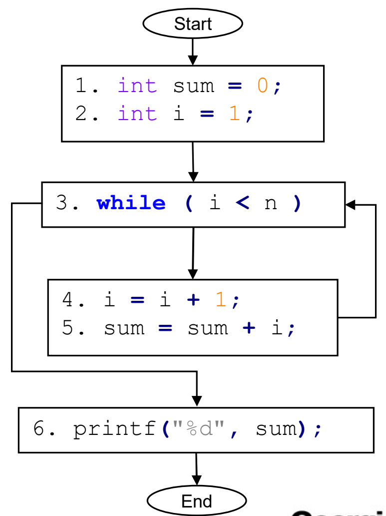
- Dynamic tracing is assumed to be lossless
  - NO processing of output during runtime!
  - All analysis & processing takes place offline
- A trace must allow offline analysis tool to faithfully recreate the analysis target
  - **Control flow tracing** --- sequence of executed statements
  - **Dependence tracing** --- sequence of exercised dependencies
  - **Value tracing** --- sequence of values that are produced by each instruction
  - **Memory access tracing** --- sequence of memory references during an execution
- Therefore, tracing is the most fine-grained of all possible dynamic analyses

Georgia Tech

50

Fine-Grained Tracing is Expensive

```
void sumUp(int n) {
    int sum = 0;
    int i = 1;
    while ( i < n ) {
        i = i + 1;
        sum = sum + i;
    }
    printf("%d",
sum);
}
```



Trace(n=6) = ?

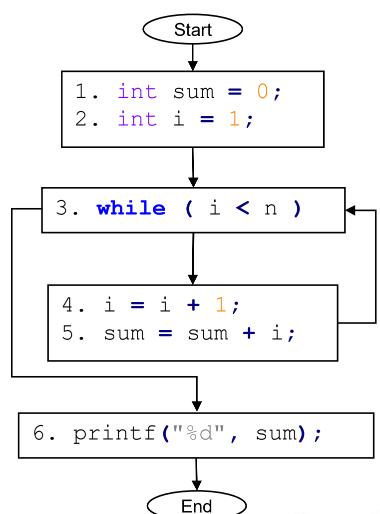
Trace(n=6): 1 2 3 4 5 3 4 5 3 4 5 3 4 5 3 4 5 3 6

Space Complexity: 4 bytes * Execution length!

51

Basic Block Level Tracing

```
void sumUp(int n) {
    int sum = 0;
    int i = 1;
    while ( i < n ) {
        i = i + 1;
        sum = sum + i;
    }
    printf("%d",
sum);
}
```



Trace (n=6) : 1 2 3 4 5 3 4 5 3 4 5 3 4 5 3 4 5 3 6

BB Trace: 1 34 34 34 34 34 36

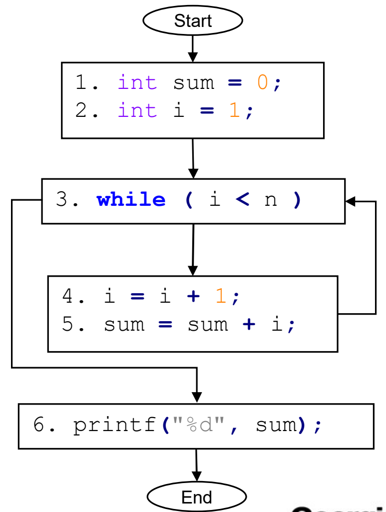
52

More Ideas

- Would function level tracing work?
  - Each trace entry is a function call & its parameters
  - No…
  - Function's behavior can be affected by global variables ☹
  - Cannot distinguish between nested functions or those called in sequence! ☹
- Predicate tracing

Fine-grained Trace	Predicate Trace
1 2 3 6	F
1 2 3 4 5 3 6	T F

- The trace is no longer randomly accessible
- Must start from the beginning to understand



53

Lesson Summary

- Discussed how to employ the tracing process for dynamic analysis
- Discussed how to Apply Static and Dynamic BinaryInstrumentation
- Discussed how to Create a Pintool to utilize PIN
- Discussed Valgrind and QEMU techniques of instrumentation

54