# Dynamic Program Analysis

Jun Ma

majun@nju.edu.cn

# Overview

## Static analysis

> A program that takes **programs** as input and produces useful results (without executing it).

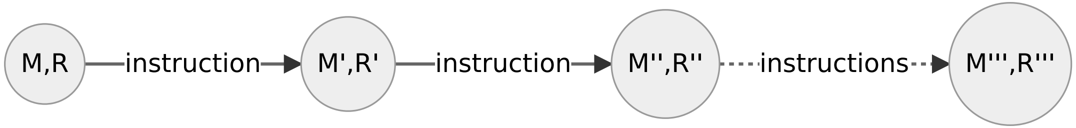
## Dynamic analysis

> A program that *monitors* and *alters* **program execution** to produce useful results.

# Computer Systems as State Machine

# Computer Systems …

> Computer system = state machine of (memory, registers) whose running is driven by instructions.
>
> (Because computer systems are simply circuits.)



This model works for

- user-level programs (`syscall` is a special non-deterministic instruction)
- operating systems (may have external interrupts)
- concurrent/multiprocessor systems (we can choose a thread for executing an instruction)

# Dynamic Analysis

> A program that monitors and alters program execution to produce useful results.

That is, a function $f(\tau)$ to produce useful results given the execution trace $\tau$ of a state machine (program/computer system).

Only provides useful results for the given $\tau$

- usually complete but unsound
  - complements static analyses
- SE tasks tolerate unsound and incomplete analyses
  - as long as results are useful in engineering
  - PL guys don’t like this

# Debuggers

# The GNU Project Debugger (GDB)

> GDB, the GNU Project debugger, allows you to see what is going on “inside” another program while it executes – or what another program was doing at the moment it crashed.

- Start your program, specifying anything that might affect its behavior.
- Make your program stop on specified conditions.
- Examine what has happened when your program has stopped.
- Change things in your program, so you can experiment with correcting the effects of one bug and go on to learn about another.

# GDB’s Offer

## Lots of commands

- Execution control `` `r, c, f, n, s, si` ``,…
- Breakpoints `` `b, hb, wa` ``, …
- Program state display `` `p, x, i, bt` ``, …
- Program state modification `` `set` ``, …
- Black magic - *reverse debugging*:
  - `` `record, rc, rn, rsi` ``, …

## Suffices for *anything*

- GDB captures the entire “state transition” procedure of a process

# Debugger is ALL Dynamic Analyses

> Any practical dynamic analysis is a “simplified” (and more efficient) debugger.

Virtually, we can do any observation or perturbation on a debugger

- Understanding program states
  - `` `info inferiors; thread 1; info registers; x/i $rip` ``
- Modifying program states
  - `` `set var = value` ``

But single-step execution incurs

- 1000X slowdown and GB/s instruction log

# Implementing GDB

## The fundamental problem:

> *How to pause program execution at an instruction (address) or statement?*

## Dynamic program instrumentation

> patch the instruction! (quite clever idea)

- make the code writable (thus cannot breakpoint on ROM addresses)
  - `` `mprotect()` ``
- patch the instruction with a “debugger trap”
  - `` `int $3` `` (`` `0xcc` `` for x86) or `` `ebreak` `` (for risc-v)
  - OS will send a signal to the parent process (gdb)
- restore the instruction after hitting the breakpoint

# Dynamic Analyses in SE Research

# Dynamic Analyses in SE Research

> How to implement ***lightweight logging*** and ***efficient analysis*** for a specific SE research task

## Problem space

- What to be analyzed?
  - Follow existing work?
  - Practical cases?

## Design space

- What to log (system design)
- How to efficiently log (hacking)
- How to analyze the logs (algorithm design)

# Example (1): Record and Replay

> We don’t need every memory/register snapshots on each instruction for a **deterministic** replay.

- E.g., `` `rr record/replay` `` provided by rr-debugger

## We only need to record **non-determinism** outcomes

- Non-deterministic instructions (e.g., RDRAND)
- #I/O (or system call)
- Timing of context switch
- Shared memory ← hard problem
  - jyy’s PhD thesis

# Example (2): Profiler

> Record even less (by **sampling**) to see which parts took the most time.

Premature optimization is the root of all evil (D. E. Knuth)

- Use profiler (**gprof**, perf/systemtap, VisualVM, …)

# Example (2): Profiler

> Record even less (by **sampling**) to see which parts took the most time.

Premature optimization is the root of all evil (D. E. Knuth)

- Use profiler (gprof, perf/systemtap, VisualVM, …)
- How to implement?
  - place a lot of “probes” in the code
    - function call, system call, interrupt, …
    - you can implement a profiler in your OSLab!
  - record time stamp and some statistics

# Example (3): Program Comprehension

## Invariant Mining

- Daikon reports *likely* invariants
  - What I see is what should happen
  - What I didn't see is what shouldn't happen

> runs a program, observes the values that the program computes, and then reports properties that were true over the observed executions.

> - Example properties
>   - x.field > abs(y);
>   - y = 2*x+3;
>   - array a is sorted;
>   - …

## Useful in many scenarios!

- Sequential programs, CSP, concurrent programs, distributed systems, …
- You may find more research opportunities: contracts, etc.

# Example (4): Bug Detection

## Online monitoring of predefined bug patterns

- AddressSanitizer (ASan)
  - memory errors: use-after-free, use-after-return, stack/heap/buffer overflow, by a shadow memory
  - Valgrind provides shadow register/memory, with better soundness (we have this paper in the reading list)
- ThreadSanitizer (TSan)
  - detects data races and deadlocks
- Hardware-assisted AddressSanitizer(HWASAN)
  - a newer variant of AddressSanitizer that consumes much less memory
- UndefinedBehaviorSanitizer (UBSan)
  - checks for other problems (e.g., signed integer overflows)

# Implementing UBSanitizer

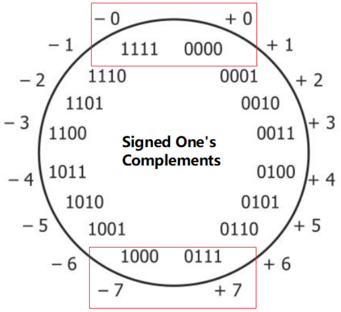
> Signed integer overflow is undefined behavior

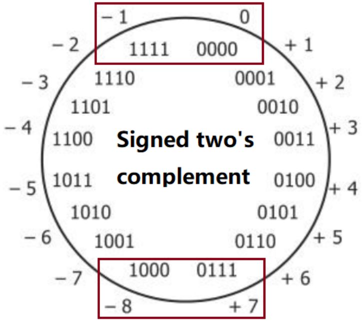
- Why?
  - to support one's complement and weird machines that throw exceptions on overflow

- How to detect them?

Add a check on each signed integer operation

- AST rewrite: `` `foo(i++, j++) + 1` `` → `` `ADD(foo(INC(i), INC(j)), 1)` ``





# Example (5): Self-Adaptive Systems

Dynamic analyses can also perturb program execution at runtime!

# Summary

# Dynamic Analysis: A Simplified Debugger

(For SE tasks.)

Implementations

- Program instrumentation
  - by changing AST/IR/ByteCode (using clang/LLVM pass/Soot/Javaassist/…)
- Dynamic instrumentation
  - by patching instructions (gdb, PIN)
- Hardware assisted
  - watch point, VM exit, PMU, PT, …