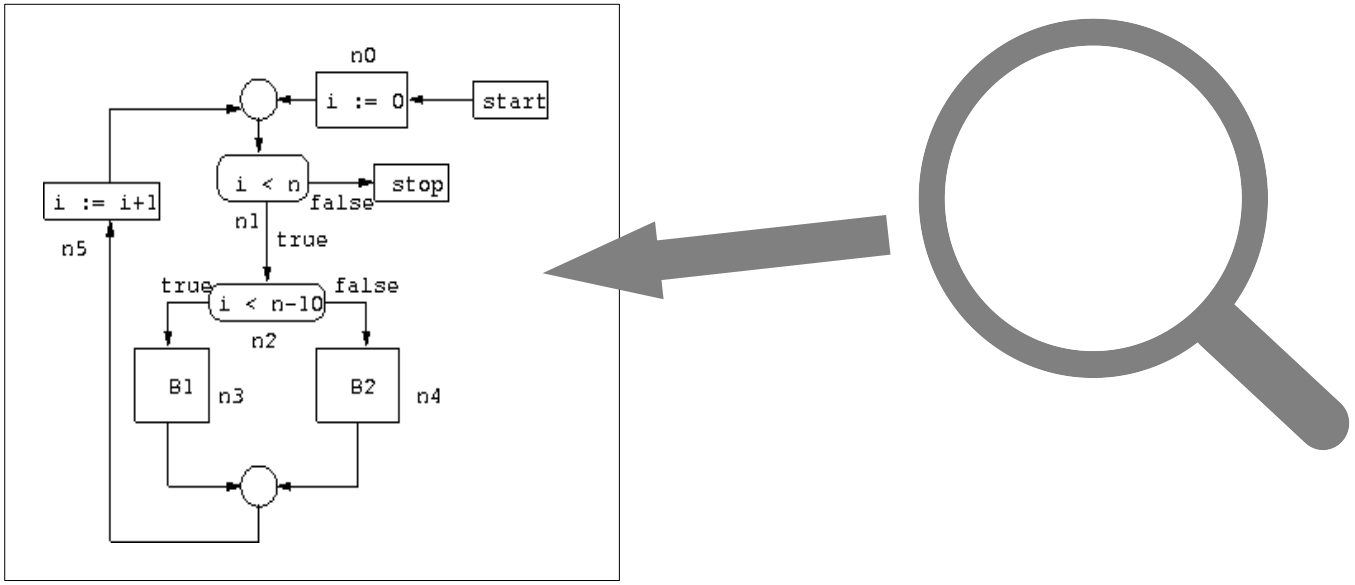
Dynamic Program Analysis
Lecture 9: Implementing Dynamic Analysis



Topics of this Lecture

- How to Implement Dynamic Analysis
- Dynamic Analysis Tools
- Wrapping Up

How to Implement Dynamic Analysis

There are basically five ways to do it:

- Source code instrumentation
- Binary rewriting
- Emulation
- Virtual machine
- Using hardware support

Source Code Instrumentation

Insert statements in the source code to collect the interesting data

Example: detecting when array access `a[i]` is out of bounds (instrumentation code in red):

```
int a[n];
...
trace_file = fopen("traces","w");
...
if (i<0 || i>n-1)
  fprintf(trace_file,"i = %d: out of range",i);
... a[i] ...
...
fclose(trace_file);
```

Source Code Instrumentation II

Source code instrumentation requires access to the source code

The instrumentation code will slow down the code a lot (typically 10-100 times)

The instrumentation can give a *probe effect*, affecting the property it aims to observe (like execution time, cache hits/misses)

On the other hand it is a straightforward way to do it, and allows for very general instrumentation

Easy to build tools

Binary rewriting

Similar to source code instrumentation, but the binary code is rewritten rather than the source code

Does not require the source code

Can be hard! Requires a thorough analysis of the binary code as to not break it when instrumenting it

Similar problems with probe effects as for source code instrumentation

Emulation/Virtual Machine

Rather than transforming the code to be run, modify the machine it runs on so it collects the interesting information

An *emulator* for the code is instrumented to do this

The code to be analyzed need not be modified

Slow! Overheads for the emulation adds to the instrumentation overheads

Can avoid probe effects by emulating the property to be observed (like execution time), but this adds to the overheads

For languages using a virtual machine (like java), the virtual machine can be instrumented instead

Using Hardware Support

Many processors have hardware support for tracing and debugging

Example: the NEXUS interface (standardized)

These interfaces can be used for dynamic analysis

Pros: no code instrumentation necessary, no probe effects, fast

Cons: Less flexible. Restricted to analyzing what the interface provides (typically a few kinds of traces, on low level)

Using Hardware Support (II)

Many modern processors have **performance counters**

Hardware registers where interesting statistics is collected

Examples: # of cache misses, # of floating point instructions executed

Different processors collect different statistics

Useful mostly to evaluate performance

Not so useful for finding logical bugs

Dynamic Analysis Tools

There are many, here are a few:

Valgrind: a framework for building dynamic analysis tools. Some examples:

- **Memcheck**: look for memory-management problems
- **Cachegrind**: cache profiler
- **Massif**: heap profiler
- **Helgrind**: thread debugger, detects data races between threads

IBM/Rational **Purify**, detects memory-related problems

Parasoft **Insure++**, memory debugging

Java PathExplorer, check that java programs adhere to specifications

Wrapping Up

Dynamic program analysis works by:

1. Setting up a mechanism for observing some aspect of a programs behaviour
2. Running the program with adequate test data, observing its behaviour and extracting the interesting information

Used for debugging, profiling, program comprehension, security vulnerability analysis

The analysis is *not safe* in general – absence of errors can not be proved (only be made likely)

Many tools exist, and are being used