EDA045F: Program Analysis

LECTURE 8: DYNAMIC ANALYSIS 1

Christoph Reichenbach

In the last lecture. . .

- More Points-to Analysis
- Memory Errors

Challenges to Static Analysis

- Static analysis is far from solved
- Very active research area
- Even with current state-of-the-art, some fundamental limitations apply
- *Bounds of computability* are only one of them...

Reflection

Java

```
Class<?> cl = Class.forName(string);
Object obj = cl.getConstructor().newInstance();
System.out.println(obj.toString());
```

- Instantiates object by string name
- Similar features to call method by name
- **Challenge**:
  - `obj` may have *any* type $\Rightarrow$ imprecision
  - Sound call graph construction very conservative
- **Approaches**
  - Dataflow: what strings flow into `string`?
    - Common: use of string prefixes
    - `Class.forName`: class only from some point in package hierarchy
    - Method calls by reflection: only methods with prefix (e.g., ("test" + ...))
  - Dynamic analysis and other approaches that we will cover later

Dynamic Loading

C

```c
handle = dlopen("module.so", RTLD_LAZY);
op = (int (*)(int)) dlsym(handle, "my_fn");
```

- Dynamic library and class loading:
  - Add new code to program that was not visible at analysis time
- **Challenge**:
  - Can't analyse what we can't see
- **Approaches**:
  - Conservative approximation
    - Tricky: External code may modify *all that it can reach*
  - Disallow dynamic loading
  - With dynamic support and annotations:
  - Allow only loading of signed/trusted code
    - signature must guarantee properties we care about
  - *Proof-carrying code*
    - Code comes with proof that we can check at run-time

Native Code

Java

```java
class A {
  public native Object op(Object arg);
}
```

- High-level language invokes code written in low-level language
  - Usually C or C++
  - May use nontrivial interface to talk to high-level language
- **Challenge**:
  - High-level language analyses don't understand low-level language
- **Approaches**:
  - Conservative approximation
    - Tricky: External code may modify *anything*
  - Manually model known native operations (e.g., Doop)
  - Multi-language analysis (e.g., Graal)

`eval` and dynamic code generation

Python

```
eval(raw_input())
```

- Execute a string as if it were part of the program
- **Challenge**:
  - Cannot predict contents of string in general
- **Approaches**:
  - Disallow `eval`
    - Not part of C, C++, Java
    - Common in dynamic languages
  - Conservative approximation
    - Tricky: code may modify *anything*
  - Dynamically re-run static analysis
  - Special-case handling (cf. reflection)

Summary

- Static program analysis faces significant challenges:
  - **Decidability** requires lack of precision or soundness for most of the interesting analyses
  - **Reflection** allows calling methods / creating objects given by arbitrary string
  - **Dynamic module loading** allows running code that the analysis couldn't inspect ahead of time
  - **Native code** allows running code written in a different language
  - **Dynamic code generation** and `eval` allow building arbitrary programs and executing them
  - No universal solution
  - Can try to 'outlaw' or restrict problematic features, depending on goal of analysis
  - Can combine with dynamic analyses

More Difficulties for Static Analysis

- Does a certain piece of code actually get executed?
- How long does it take to execute this piece of code?
- How important is this piece of code in practice?
- How well does this code collaborate with hardware devices?
  - Harddisks?
  - Networking devices?
  - *Caches* that speed up memory access?
  - *Branch predictors* that speed up conditional jumps?
  - The *ALU(s)* that perform arithmetic in the CPU?
  - The *TLB* that helps look up memory?

  ...

Impossible to predict for all practical situations

Static vs. Dynamic Program Analyses

	Static Analysis	Dynamic Analysis
Principle	Analyse program structure	Analyse program execution
Input	Independent	Depends on input
Hardware/OS	Independent	Depends on hardware and OS
Perspective	Sees everything	Sees that which actually happens
Soundness	Possible	Must try all possible inputs
Precision	Possible	Always, for free

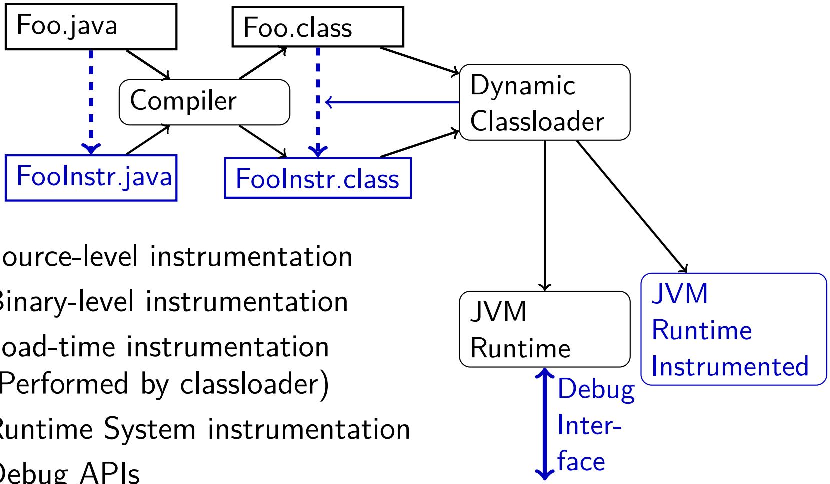
Summary

- Static analyses have known limitations
- Static analysis cannot reliably predict dynamic properties:
  - How often does something happen?
  - How long does something take?
- This limits:
  - Optimisation: *which optimisations are worthwhile?*
  - Bug search: *which potential bugs are 'real'?*
- Can use *dynamic analysis* to examine run-time behaviour

Gathering Dynamic Data

- **Instrumentation**
- Performance Counters
- Emulation

Gathering Dynamic Data: Java



- Source-level instrumentation
- Binary-level instrumentation
- Load-time instrumentation (Performed by classloader)
- Runtime System instrumentation
- Debug APIs

Comparison of Approaches

- **Source-level instrumentation**:
  - + Flexible
  - − Must handle syntactic issues, name capture, ...
  - − Only applicable if we have all source code
- **Binary-level instrumentation**:
  - + Flexible
  - − Must handle binary encoding issues
  - − Only applicable if we know what binary code is used
- **Load-time instrumentation**:
  - + Flexible
  - + Can handle even unknown code
  - − Requires run-time support, may clash with custom loaders
- **Runtime system instrumentation**:
  - + Flexible
  - + Can see everything (gc, JIT, ...)
  - − Labour-intensive and error-prone
  - − Becomes obsolete quickly as runtime evolves
- **Debug APIs**:
  - + Typically easy to use and efficient
  - − Limited capabilities

Instrumentation Tools

	C/C++ (Linux)	**Java**
Source-Level	C preprocessor	ExtendJ
Binary Level	`pin`, `llvm`	`soot`, `asm`, `bcel`, AspectJ
Load-time	?	Classloader, AspectJ
Debug APIs	`strace`	JVMTI

- Low-level data gathering:
  - Command line: `perf`
  - Time: `clock_gettime()` / `System.nanoTime()`
  - Process statistics: `getrusage()`
  - Hardware performance counters: PAPI

Practical Challenges in Instrumentation

- *Measuring*:
  - Need access to relevant data (e.g., Java: source code can't access JIT)
- *Representing (optional)*:
  - Store data in memory until it can be emitted (optional)
  - May use memory, execution time, perturb measurements
- *Emitting*:
  - Write measurements out for further processing
  - May use memory, execution time, perturb measurements

Summary

- Different **instrumentation strategies**:
  - Instrument **source code** or **binaries**
  - Instrument **statically** or **dynamically**
  - Instrument **input program** or **runtime system**
- Challenges when handling analysis:
  - **In-memory representation of measurements** (for compression or speed)
  - **Emitting measurements**

Instrumentation with AspectJ

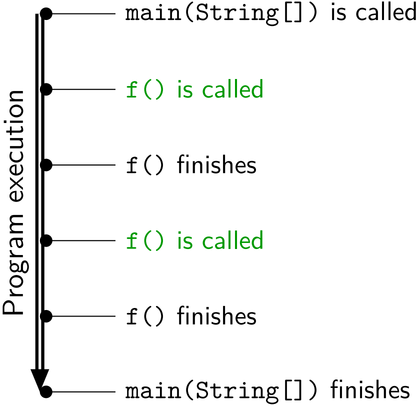
- *AspectJ* is Java tool for *Aspect-Oriented Programming*
  - Premise: separate program into different 'aspects'
  - 'weave' aspects together

$\Longrightarrow$ for analysis, weaving = instrumentation

- AspectJ permits:
  - Binary instrumentation
  - Load-time instrumentation (if supported by the target application)

AspectJ View of the World

Join Points | **Pointcut**



call f()

Pointcuts and Join Points

- *Join Point*: 'point of interest' during program execution
  - Properties of program execution
  - Method / constructor called
  - Method / constructor returns
  - Exception raised
- *Pointcut*: 'Set of join points that we are interested in'
  - Static description that captures set of dynamic events
  - Call / return to/from method/constructor of particular name / in particular class
  - Exception of a given name is raised
  - Parameters have a particular type
  - Currently executing in a particular class
  - Within another pointcut

  ...

Pointcut Examples

- `call(void se.lth.MyClass.method(int, float))`:
  Method is called
- `call(* se.lth.MyClass.method(int, float))`:
  Method is called (any return type)
- `call(private * se.lth.MyClass.*())`:
  Any private method with no arguments is called
- `call(void se.lth.MyClass.new(..))`:
  Any of the class constructors is called (overloaded)
- `execution(void se.lth.MyClass.method(int, float))`:
  Method starts
- `handler(InvalidArgumentException)`:
  Exception handler invoked
- `this(java.lang.String)`:
  'this' object is of a given type
- `target(se.lth.MyClass)`:
  Method invocation target is of the given type

Defining Pointcuts

- To work with pointcuts, we must name them
- Can introduce parameters that we can reason about later

```
pointcut testEquality(Point p):
        target(Point) &&
        args(p) &&
        call(boolean equals(Object));
```

Advice

- *Advice* is code added to a pointcut
  - Before
  - After
  - Around (may call join point multiple times or skip pointcut)
- Any regular Java code permitted
- Can access information about join point:
  - `thisJoinPoint`: Join point actual parameters, method call target
  - `thisJoinPointStaticPart`: Program location

AspectJ Example

```
import java.util.*;

public aspect Instr {

  pointcut anycall(java.lang.Object obj) :
          (call(* *(..))  && this(obj));

  static boolean trace = true;

  before(Object obj) :  anycall(obj) {
    if (trace) {
      trace = false;
      System.out.println("Calling from " + obj);
      trace = true;
    }
  }
}
```

Make sure to avoid accidental infinite recursion!

Summary

- **AspectJ** allows instrumenting Java code by:
  - Static re-writing
  - Load-time re-writing
- Allows executing code in the context of **join points**
- Join points are abstractly described through **pointcuts**
- Pointcuts are given **advice**, which is Java code
  - Advice is executed whenever join point matches pointcut
  - Can be before / after / around join points

General Data Collection

- *Events*: When we measure
- *Characteristics*: What we measure
- *Measurements*: Individual observations
- *Samples*: Collections of measurements

Events

- Subroutine call
- Subroutine return
- Memory access (read or write or either)
- System call
- Page fault

...

Characteristics

- *Value*: What is the type / numeric value / ...?
- *Counts*: How often does this event happen?
- *Wallclock times*: How long does one event take to finish, end-to-end?

Derived properties:

- *Frequencies*: How often does this happen
  - Per run
  - Per time interval
  - Per occurrence of another event
- *Relative execution times*: How long does this take
  - As fraction of the total run-time
  - As fraction of some surrounding event

Perturbation

Example challenge: can we use total counts to decide *whether* to optimise some function `f`?

- On each method entry: get current time
- On each method exit: get current time again, update aggregate
- Reading timer takes: $\sim$ 80 cycles
- Short `f` calls may be much faster than 160 cycles
- Also: measurement needs CPU registers
  $\Rightarrow$ may require registers
  $\Rightarrow$ may slow down code further

Measurements perturb our results, slow down execution

Sampling

Alternative to full counts: *Sampling*

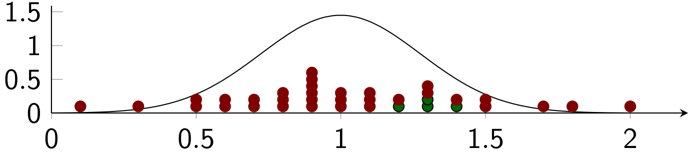
- Periodically interrupt program and measure
- Problem: how to pick the right period?

1. System events (e.g., GC trigger or safepoint)
   System events may bias results
2. Timer events: periodic intervals
    - May also bias results for periodic applications
    - Randomised intervals can avoid bias
    - Short intervals: perturbation, slowdown
    - Long intervals: imprecision

Samples and Measurements

Samples are *collections of measurements*

- Bigger samples:
  - Typically give more precise answers
  - May take longer to collect
- Challenge: representative sampling



Carefully choose what and how to sample

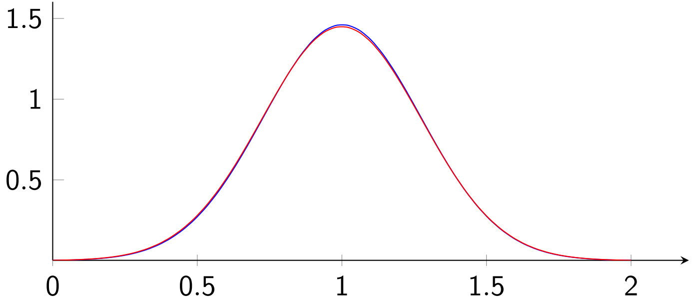
Summary

- We measure *Characteristics* of *Events*
- *Sample*: set of *Measurements* (of characteristics of events)
- Measurements often cause *perturbation*:
  - Measuring disturbs characteristics
  - Not relevant for all measurements
  - Measuring time: more relevant the smaller our time intervals get
- Can measure by:
  - Counting: observe every event
    - Gets all events
    - Maximum measurement perturbation
  - Sampling: periodically measure
    - Misses some events
    - Reduces perturbation

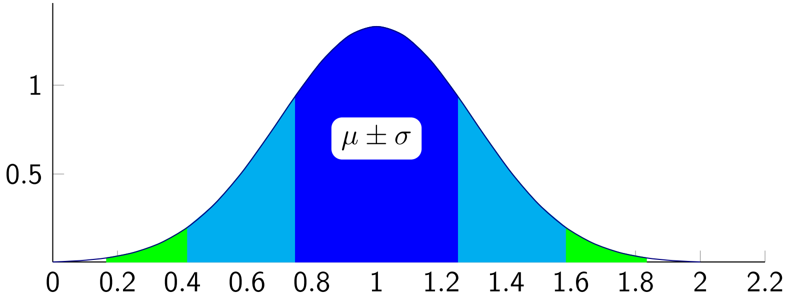
Presenting Measurements

	P1	**P2**
Mean μ	1,001	0,999
Standard Deviation σ	0,273	0,275

Assuming normal distribution:



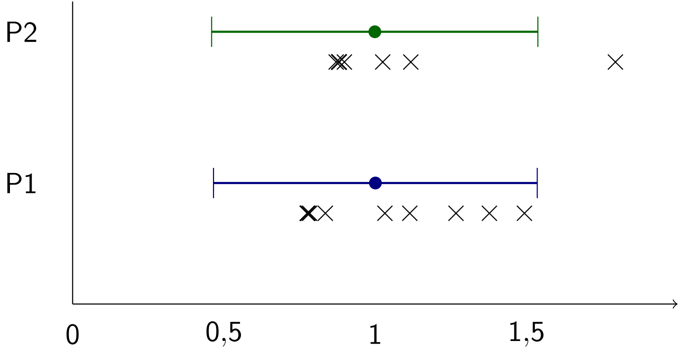
Standard Deviation, Assuming Normal Distribution



Deviation	**Chance of actual μ being in interval**
σ	68,27%
$1{,}96\sigma$	95,00%
2σ	95,45%
$2{,}58\sigma$	99,00%
3σ	99,73%

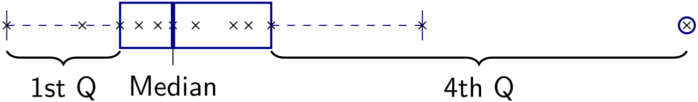
How Well Does Normal Distribution Fit?

Representation with error bars (95% confidence interval):



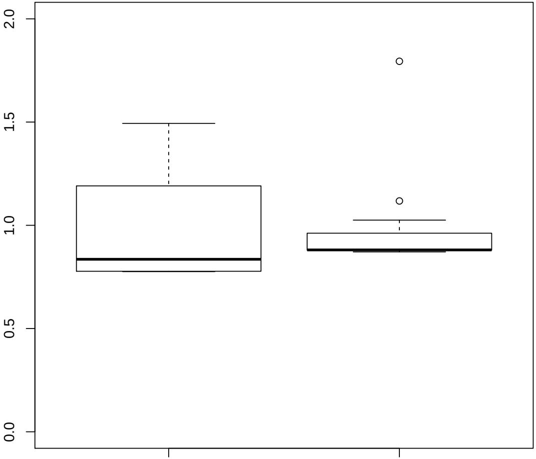
Mean + Std.Dev. are misleading if measurements don't observe normal distribution!

Box Plots

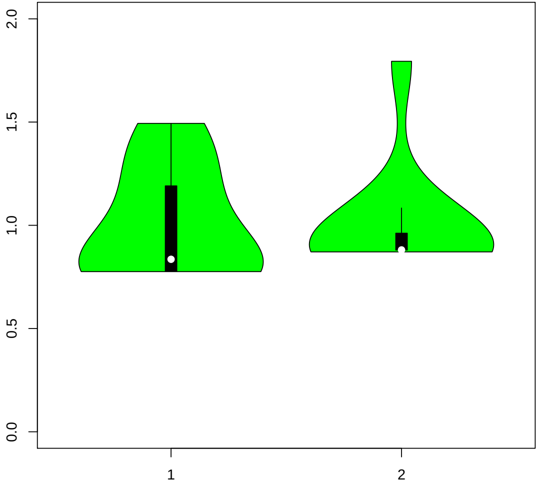


- Split data into 4 *Quartiles*:
  - Upper Quartile (1st Q): Largest 25% of measurements
  - Lower Quartile (4th Q): Smallest 25% of measurements
  - Median: measured value, middle of sorted list of measurements
- Box: Between 1st/4th quartile boundaries
  Box width = inter-quartile range (*IQR*)
- 1st Q whisker shows largest measured value $\leq$ 1,5 $\times$ IQR (from box)
- 4th Q whister analogously
- Remaining *outliers* are marked

Box plot: example



Violin Plots



Summary

- We don't usually know our statistical distribution
- There exist statistical methods to work precisely with confidence intervals, given certain assumptions about the distribution (not covered here)
- Visualising without statistical analysis:
  - **Box Plot**
    - Splits data into **quartiles**
    - Highlights points of interest
    - No assumption about distribution
  - **Violin Plot**
    - Includes Box Plot data
    - Tries to approximate probability distribution function visually
    - Can help to identify actual distribution

Homework #4

1. Use AspectJ for profiling
2. Use `perf` to analyse hardware performance counters
3. Use Soot to build a dynamic callgraph and compare it to Soot's static call graph

Review

- Basic dynamic program analysis
- Instrumentation
- Sampling

To be continued...

- More Dynamic Program Analysis